Welcome to 15-740!

15-740 FALL'19

NATHAN BECKMANN

Topics

What is computer architecture?

Underlying technology

History (of x86)

ISA design

Information about the class

The most important thing

I WELCOME YOUR INTERRUPTIONS!

An interactive class is better for everyone…

We will all learn more and have more fun!

(also it helps your grade…)

What is computer architecture?

Hot take: The foundational discipline of computer science

Before you can have “computer science”, you first need (at least a design of) a computer!

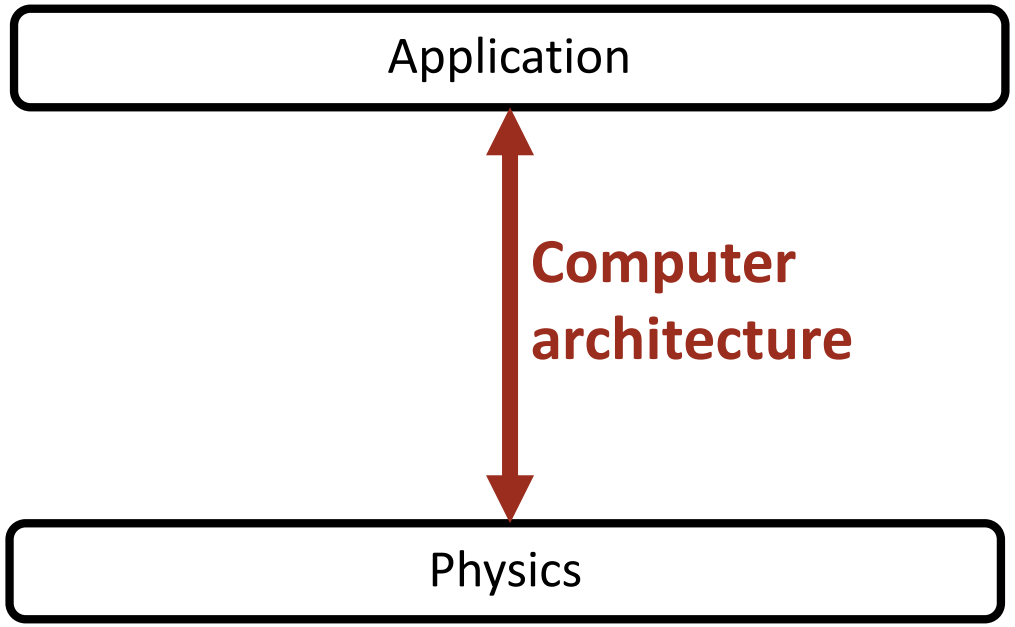
We learned a lot once we had a computer

- Programming is hard ← *Wasn’t obvious a priori!*
- Algorithmic complexity
- System building principles
- Abstractions between layers (e.g., ISAs)
- …

What is computer architecture?

The science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. [wikipedia]

Abstractions to bridge gap



Responsive to technology

- Underlying components:
  - Relays → Tubes → Transistors → VLSI → *??? Carbon nano tubes ???*
  - Mercury delay lines → Magnetic core → DRAM → FLASH → *??? Resistive RAM ???*

- What to optimize for:
  - Transistors
  - Memory
  - Instructions
  - Performance
  - Power
  - Parallelism

- **Technology constantly changing!**

Responsive to applications

So what is computer architecture?

The answer is constantly changing!

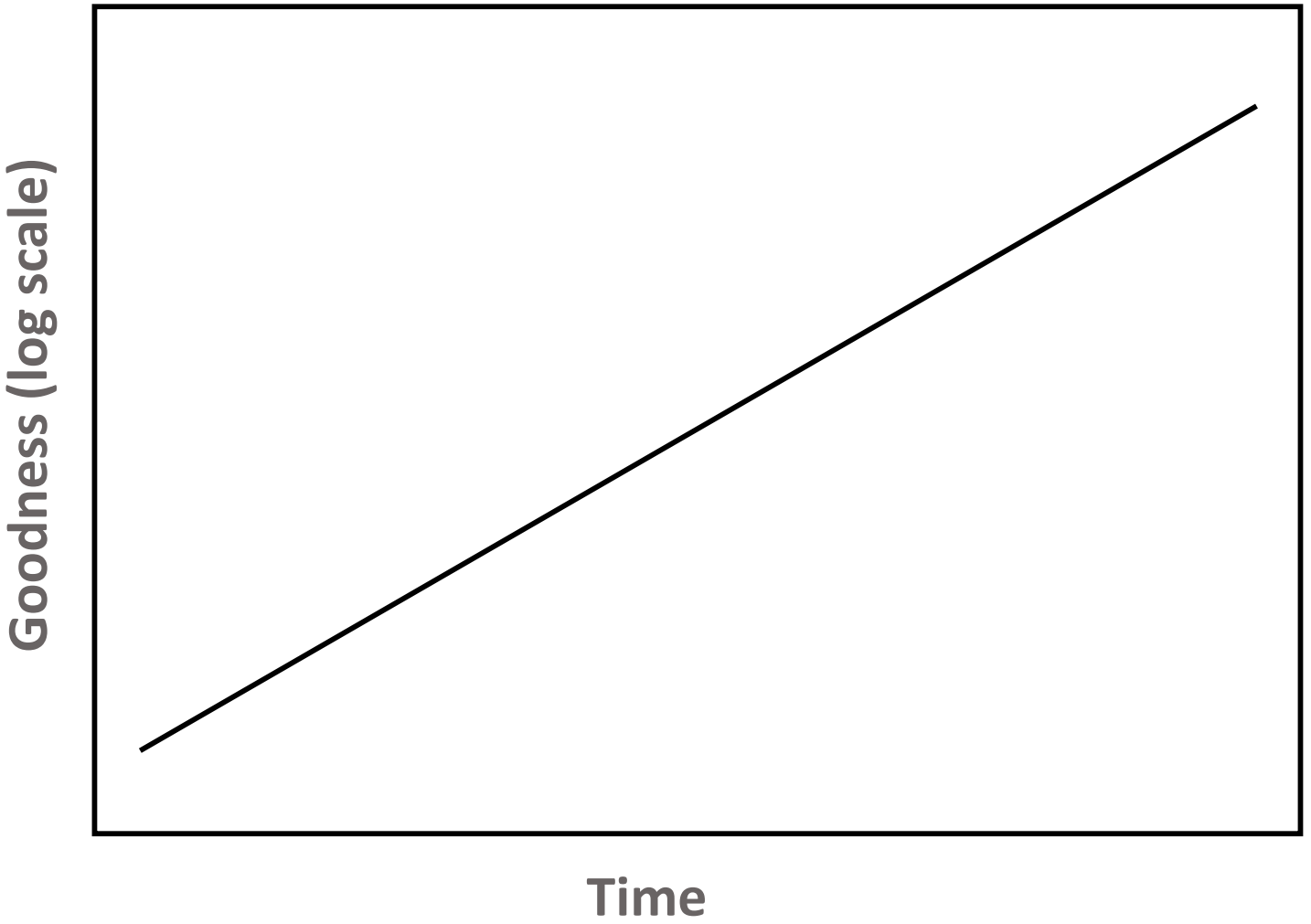
As technology and application space change, so too the focus of computer architecture…

- 1950s-60s: Computer arithmetic
- 1970s-80s: Instruction set architecture
- 1980s-90s: CPU design
- 1990s-2000s: Memory system, I/O, networks
- 2000s-today: Power, multicore
- 2010s-today: Specialized accelerators

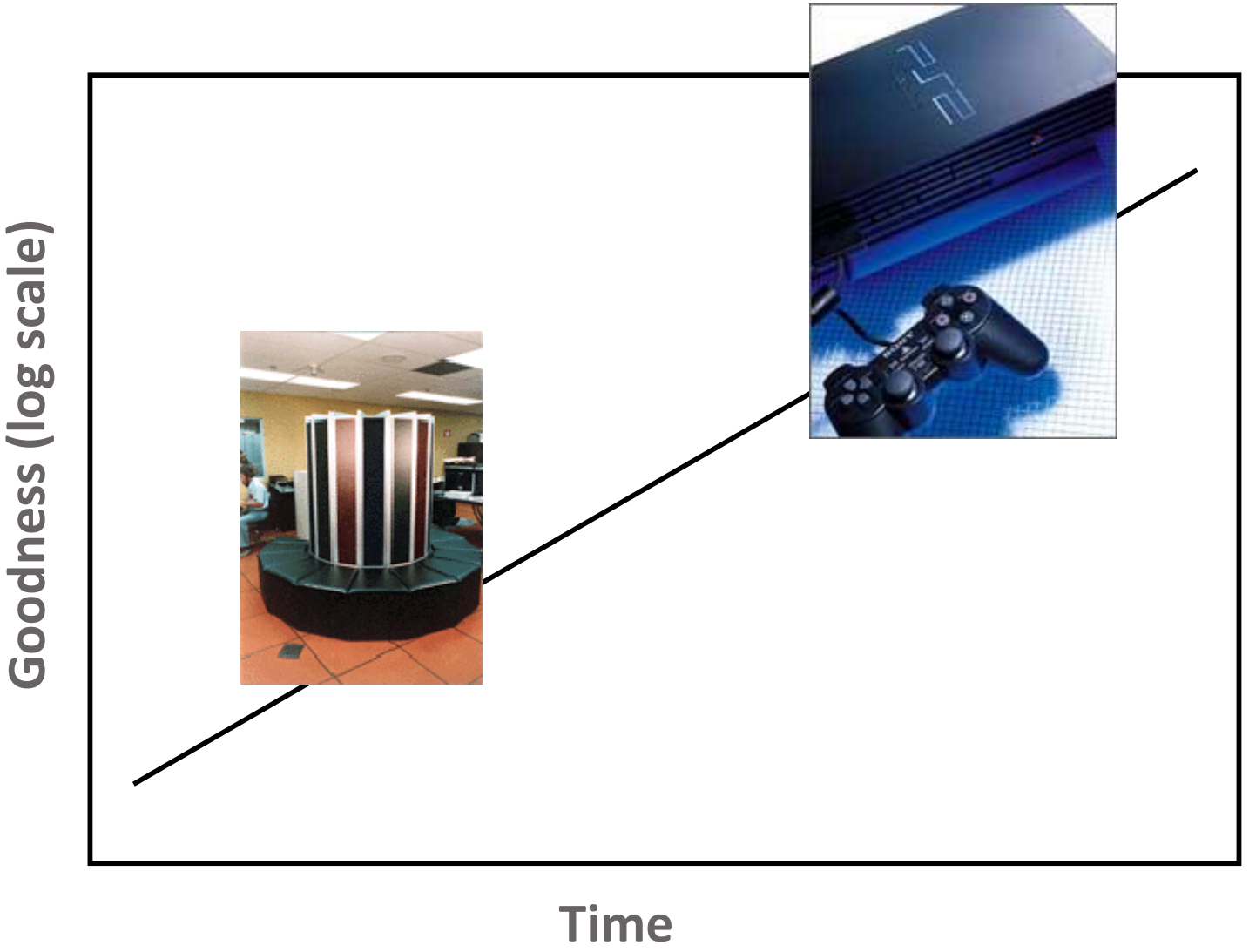
Ever-Changing Technology

50 YEARS IN 15 SLIDES

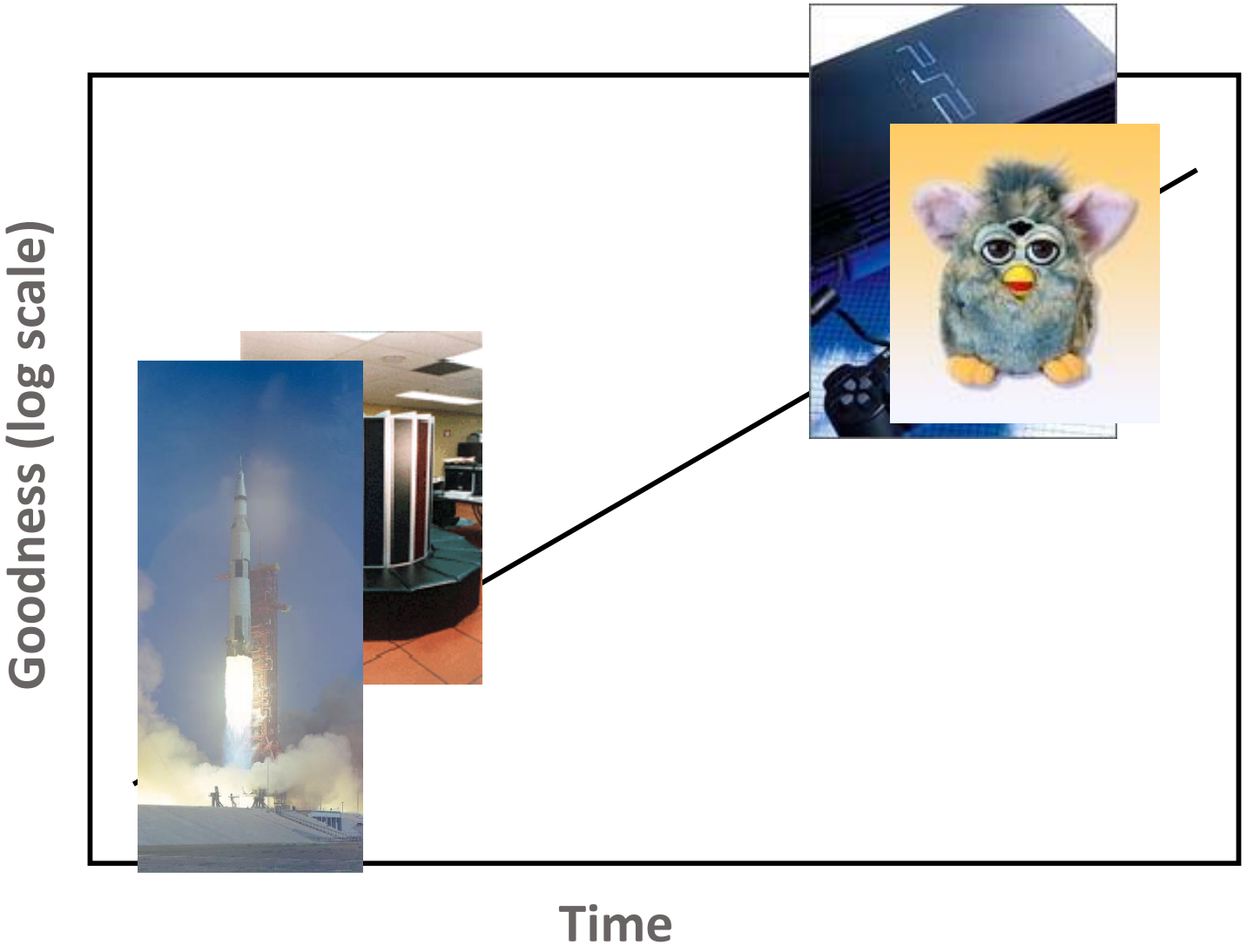
Moore’s Law



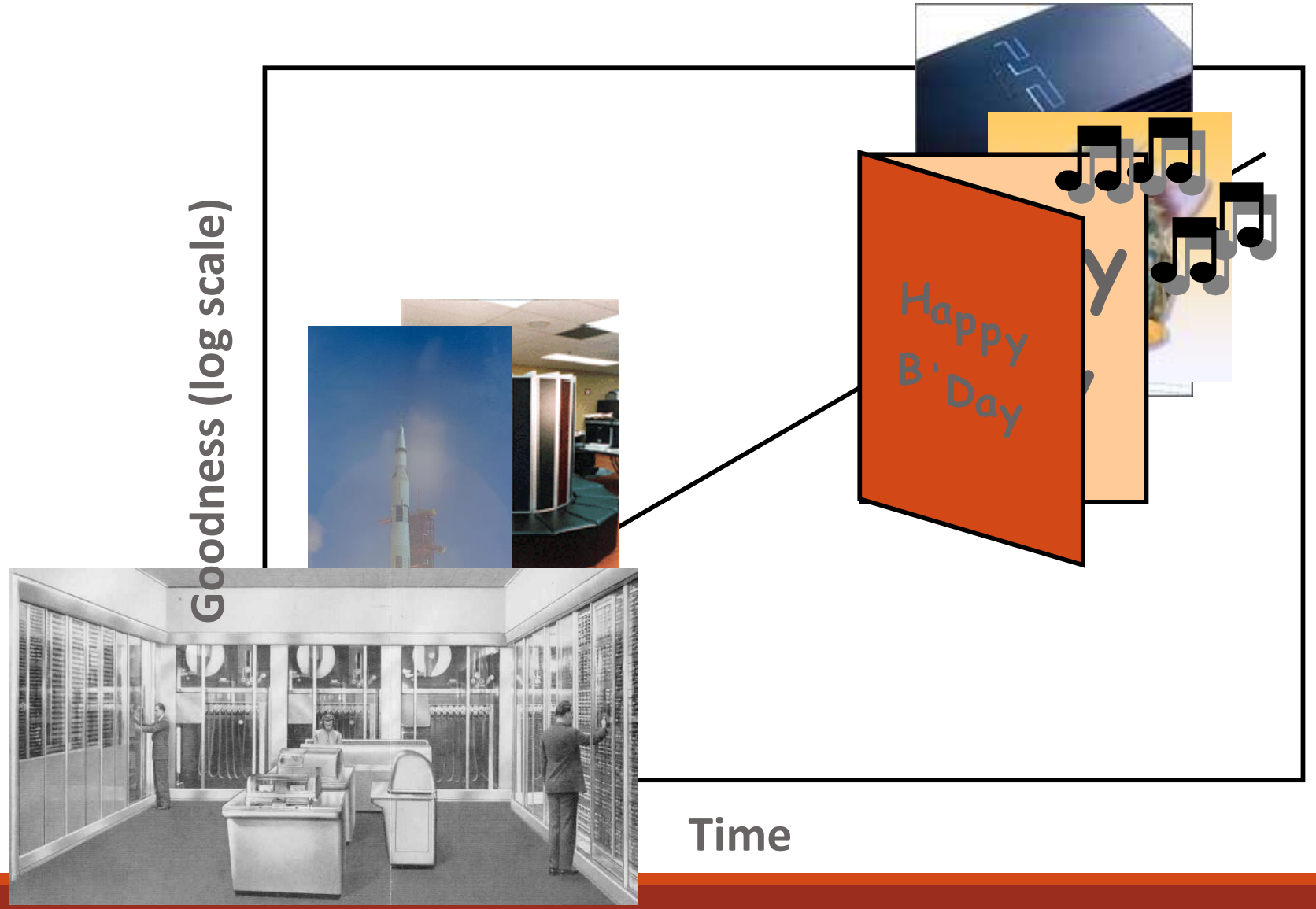
Moore’s Law



Moore’s Law

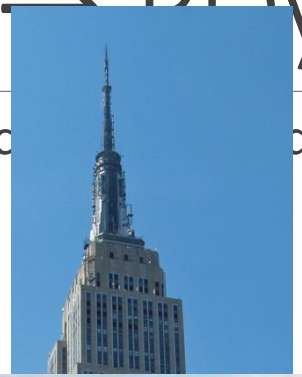


Moore’s Law



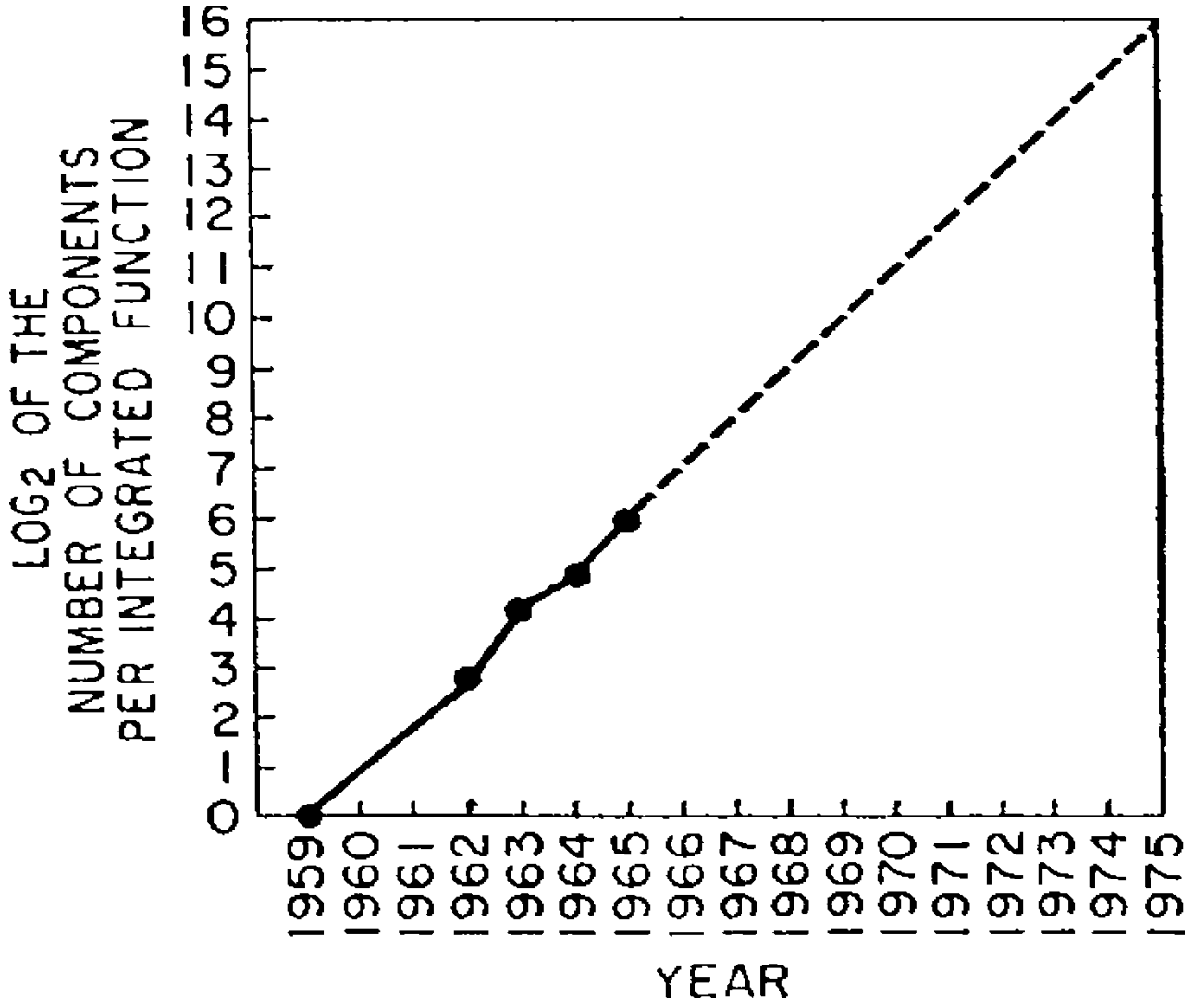
Case study: Eniac → Playstation 4

How much would enough Eniac [illegible] qual 2.8Kg of PS/4 computing?

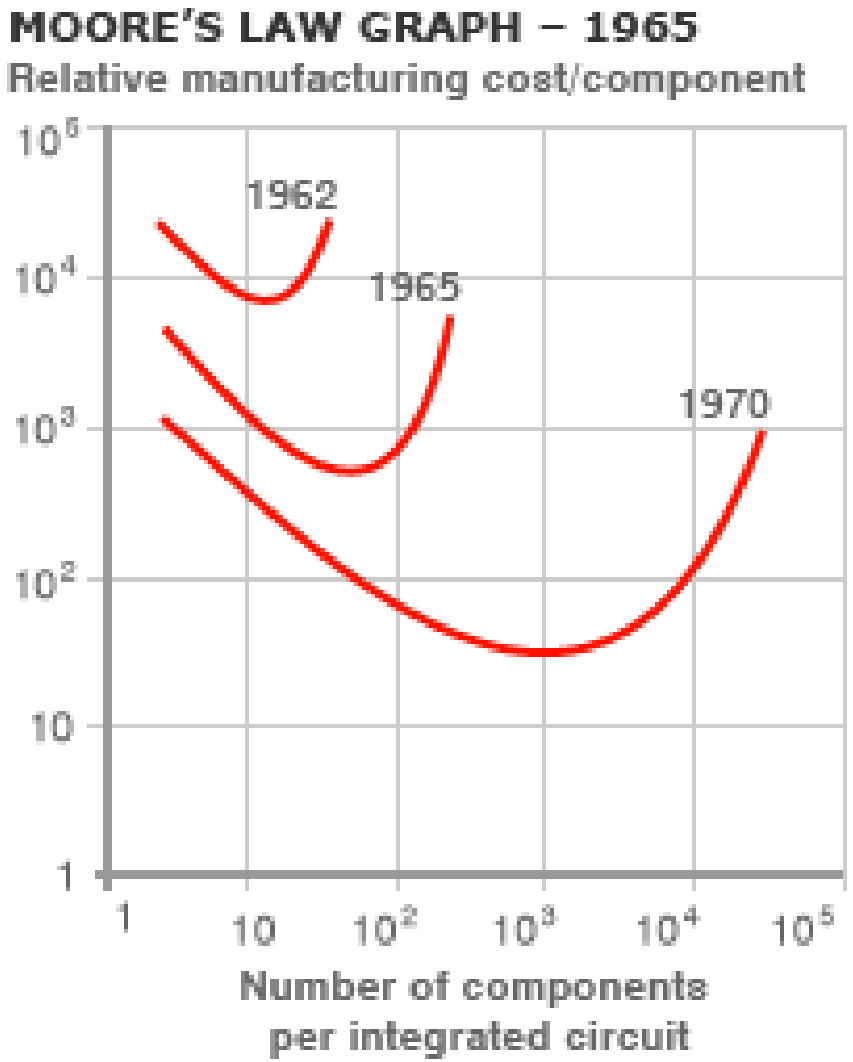


Alternatively, more than all the buildings in Pittsburgh!

Moore’s original prediction from 1965



Moore’s law really about economics?



Technology changes architecture

It isn't just transistor density

- Transistor size, density, speed, power, cost
- Memory size, density, latency, throughput
- Disks
- Networks
- Communication

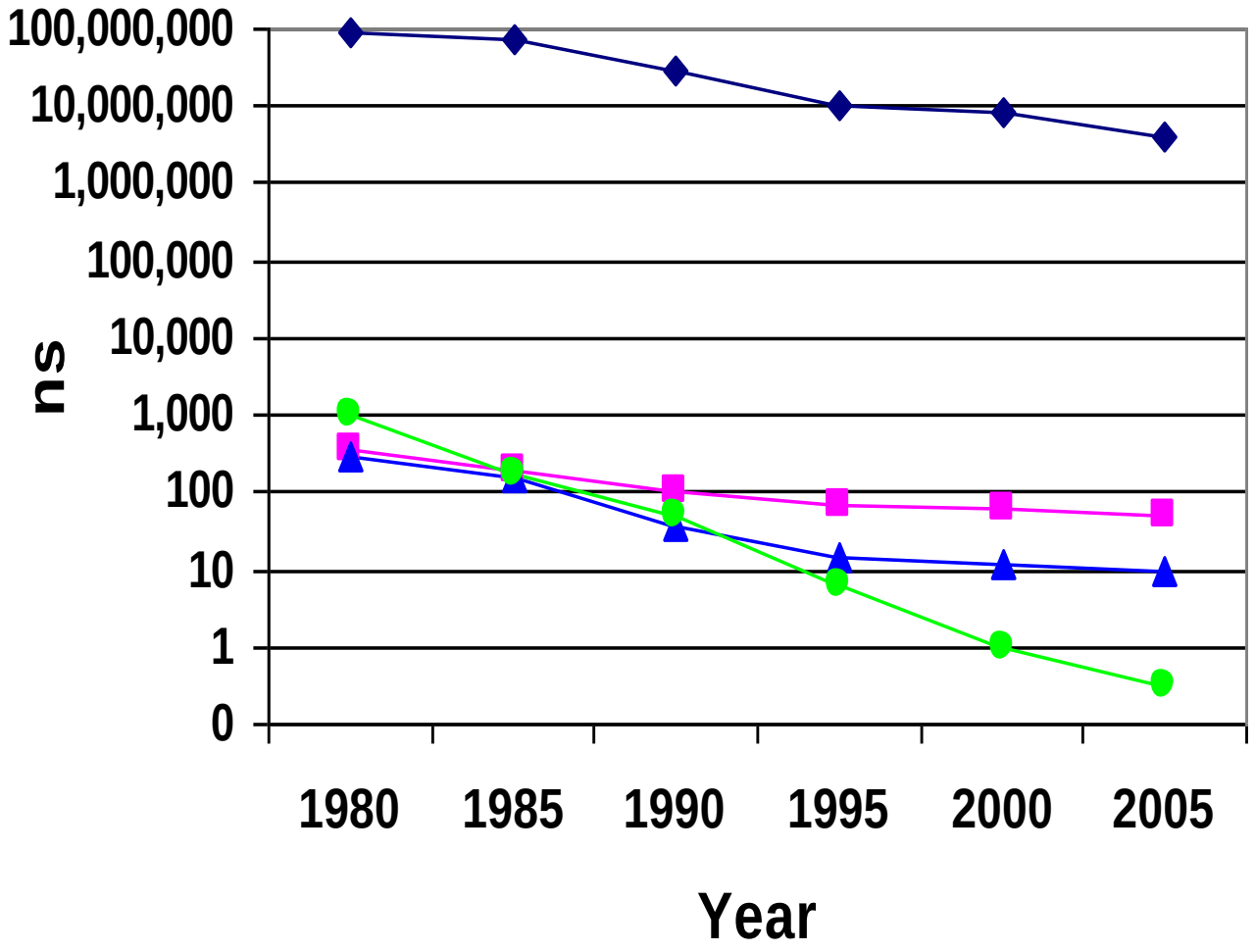
These trends lead to exponential increase in ops/sec-$-$m^3$-watt

Which in turn leads to changes in applications

- Mainframes → Desktops → Mobile

Which leads to new design goals

Case study: The CPU-memory gap



Q: Why isn't memory getting faster?

A: Exponential growth in memory **size.**
➔ *It's not all about performance!*

When did architects optimize multiplies?

When did architects optimize loads?

➔ **Technology has a dramatic impact on what's important in architecture.**

Technology constantly on the move!

Not optimizing for # transistors anymore

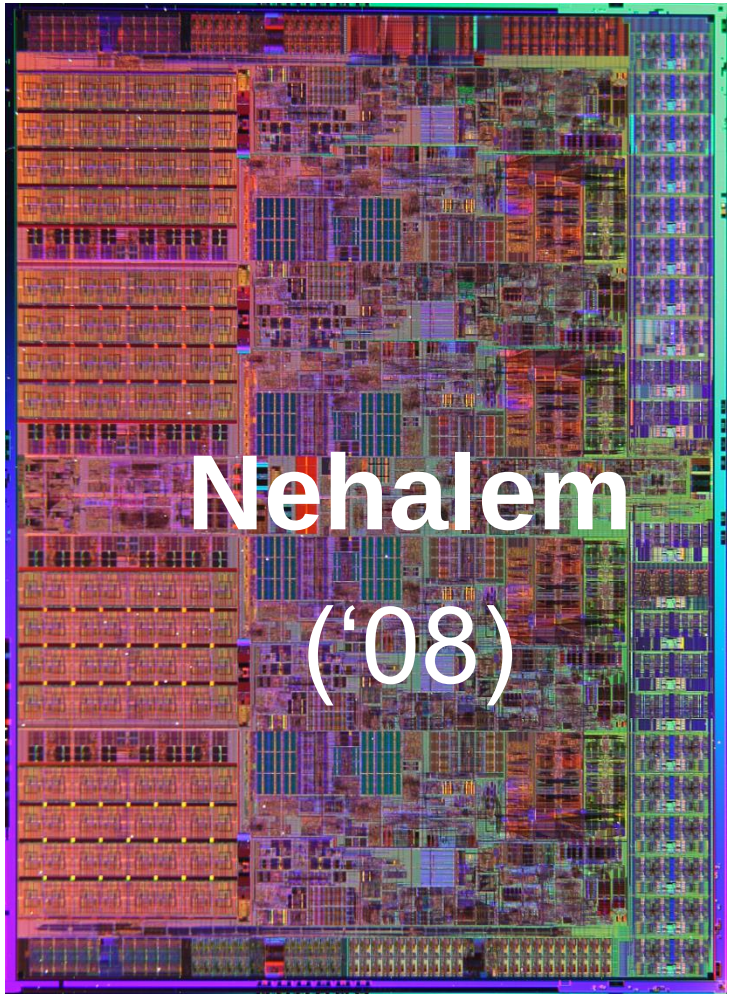
- > 1 billion transistors/chip

Issues:

- Complexity
- Power
- Heat
- Latency
- Parallelism

Huge change in thinking

- Improve sequential vs. parallel performance?
- Improve throughput vs. decrease power?
- Specialized vs. general purpose?



Why **you** should study computer architecture

Understand how computers work

Its not just how to build them:

- Why does my program run slowly?
- How do I increase performance?
- How do I improve reliability?
- Is my system secure?
- What can I expect tomorrow?

We are at a crossroads…

Case study: Deep learning

"Deep learning" (a.k.a. neural networks) are taking over the world

…An old technique that had fallen out of favor for decades

What happened?

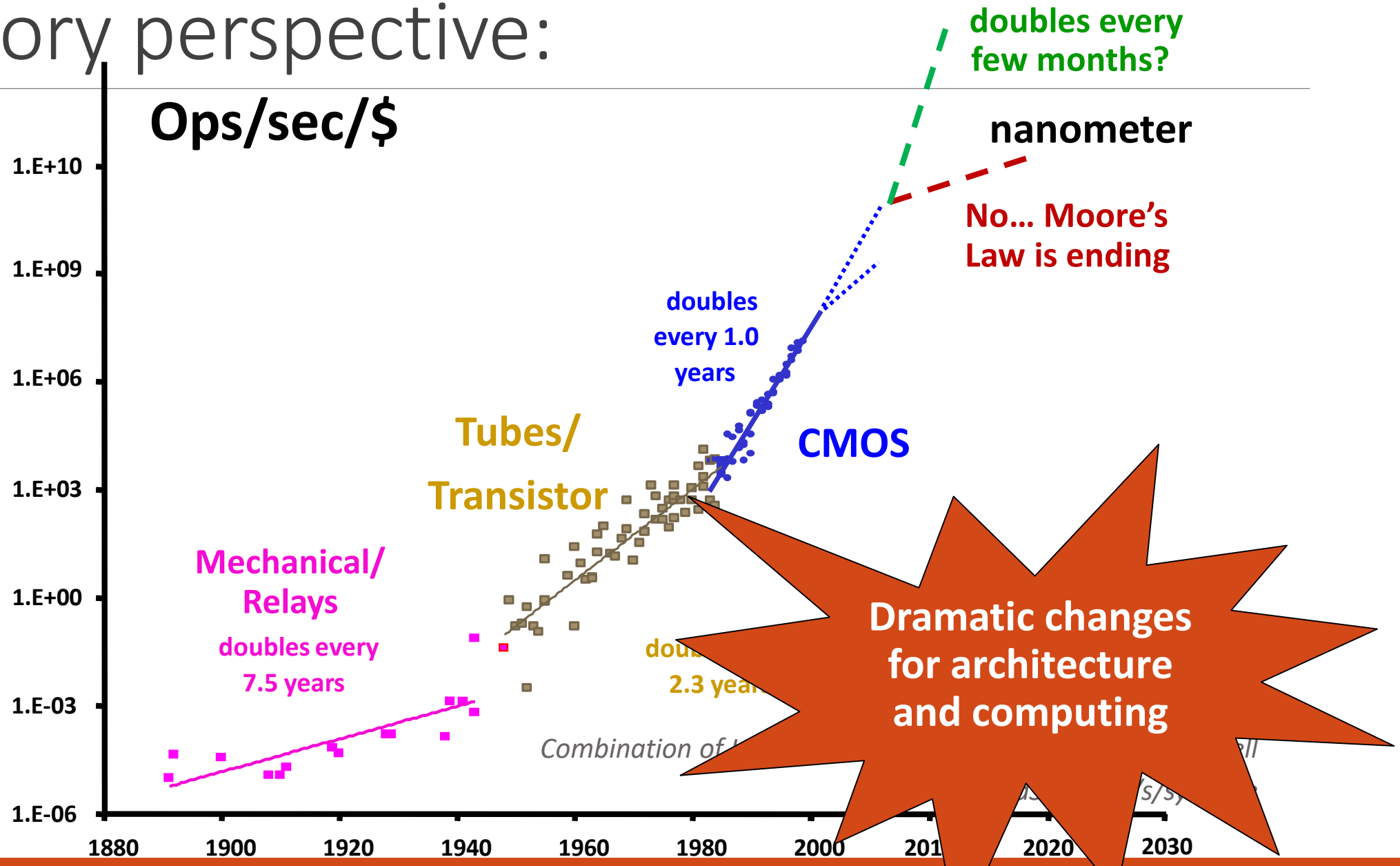
- 1) Big data – massive training datasets
- 2) GPUs – **massive compute** available for **little $$**

Now, "neural accelerators" are the hottest topic in computer architecture

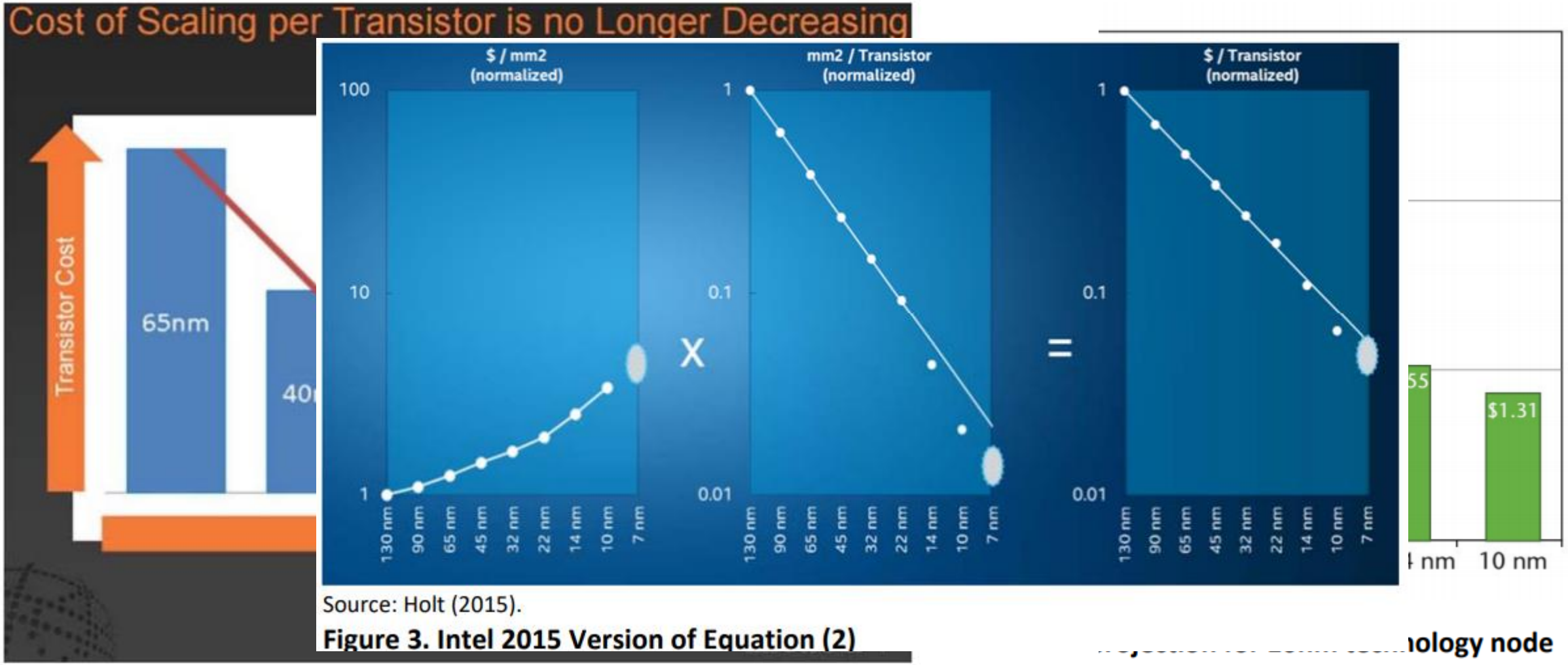
- E.g., ~one-third of papers at top arch conferences since 2016
- Google, Apple, Microsoft, 100s of startups building & deploying custom hardware

Highly specialized architectures disrupt the computational model we've all grown up with

History perspective:



Moore's Law already finished?



Source: Holt (2015).

Figure 3. Intel 2015 Version of Equation (2)

Figure 5. Global Foundries' Transistor Manufacturing Cost at Recent Technology Nodes

Source: McCann (2015).

A Brief History of Computer Architecture

The microprocessor

Microprocessor revolution

- Technology threshold crossed in 1970s:
  Enough transistors (~25K) to fit a 16-bit processor on one chip
- Huge performance advantages: fewer slow chip-crossings
- Even bigger cost advantages: one “stamped-out” component

Created new applications

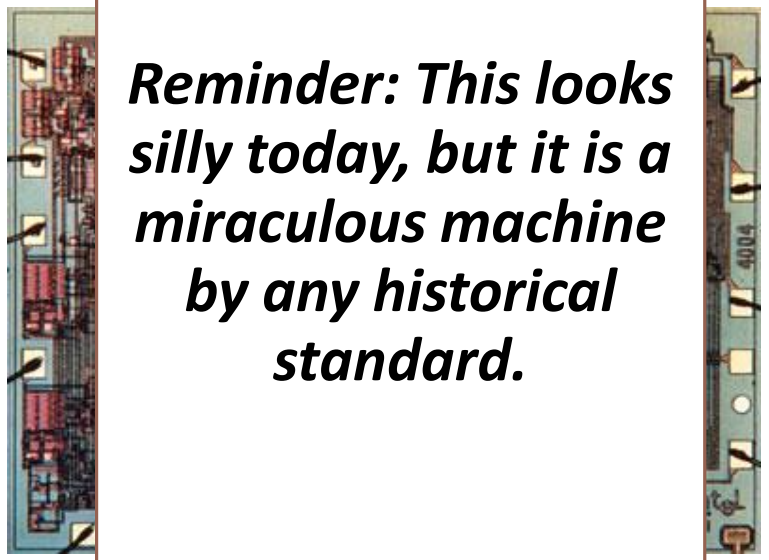
- Desktops, CD/DVD players, laptops, game consoles, set-top boxes, mobile phones, digital camera, mp3 players, GPS, automotive, …

And replaced incumbents in existing segments

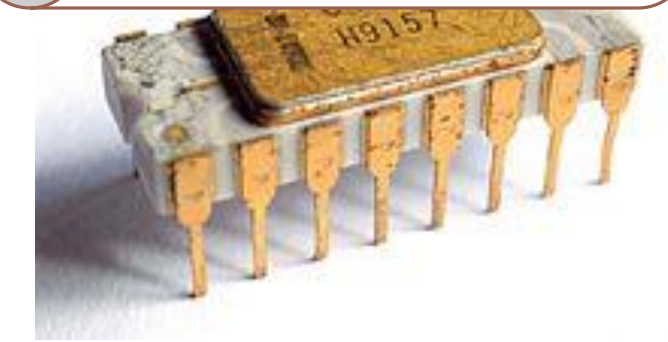
- Supercomputers, “mainframes”, “minicomputers”, etc.

First microprocessor

- Intel 4004 (1971)
  - The first single-chip CPU!
  - Application: calculators
  - Technology: 10000 nm
  - 2300 transistors in 13 mm^2
  - 740 KHz, 8 or 16 cycles/instr.
    - Multiple cycles to xfer data
  - 12 Volts
  - 640-byte address space
  - 4-bit data



Reminder: This looks silly today, but it is a miraculous machine by any historical standard.



Tracing the microprocessor revolution

- How were growing transistor counts used?

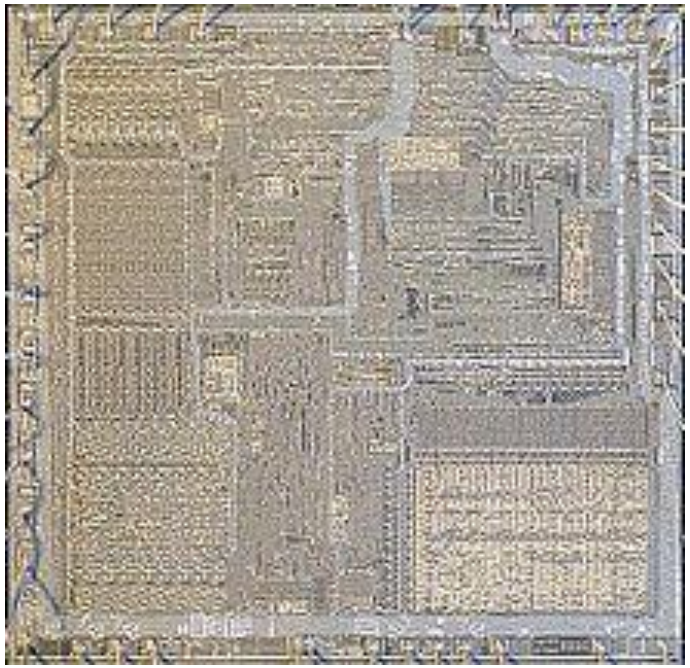
- Initially to widen the datapath
  - 4004: 4 bits → Pentium4: 64 bits

- And to add more powerful instructions
  - To amortize overhead of fetch and decode
  - To simplify programming (which was done by hand then)
  - To reduce memory requirements for program

  - Could get absurd: e.g., VAX “POLY” instruction

The first x86

- Intel 8086 (1978)
  - Application: microcomputers
  - Technology: 3000nm
  - 29,000 transistors in 33 mm^2
  - 5-10 MHz, 2-190 cycles/instr.
    - What took 190 cycles?
  - 5 Volts
  - 1MB address space
  - 16-bit datapath
  - *Microcoded* design: each instruction invokes a *microprogram* with architecture-specific micro-instructions
    - *Idea from MIT Whirlwind in 1950s!*



Implicit parallelism

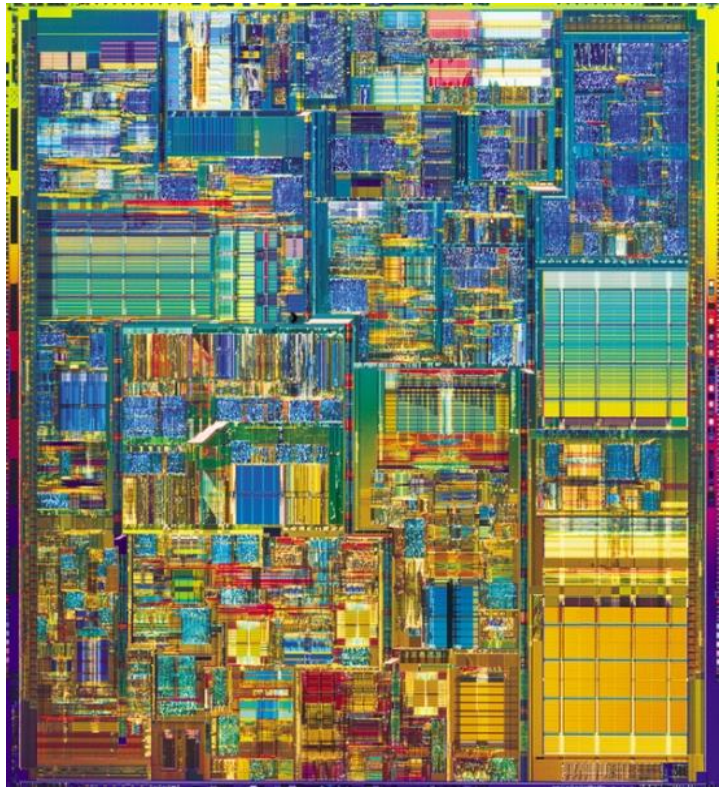
- Then to **extract implicit instruction-level parallelism (ILP)**
  - Hardware provides parallel resources, figures out how to use them
  - Software is oblivious – *for the most part!*

- Initially using pipelining …
  - Which also enabled increased clock frequency
- … caches …
  - Which became necessary as processor clock frequency increased
- … deeper pipelines and branch speculation
- … multiple instructions per cycle (superscalar)
- … dynamic scheduling (out-of-order execution)

- Meanwhile, also continued to add features, e.g., integrated floating point

Nearing the end of uniprocessors

- Intel Pentium4 (2003)
  - Application: desktop/server
  - Technology: 90nm *(1% of 4004)*
  - 55M transistors *(20,000x)*
  - 101 mm^2 *(10x)*
  - 3.4 GHz *(10,000x)*
  - **3 instrs / cycle** *(superscalar)*
  - 1.2 Volts *(1/10x)*
  - 32/64-bit data *(16x)*
  - 22-stage pipelined datapath
  - Two levels of on-chip cache
  - Data-parallel “vector” (SIMD) instructions, hyperthreading

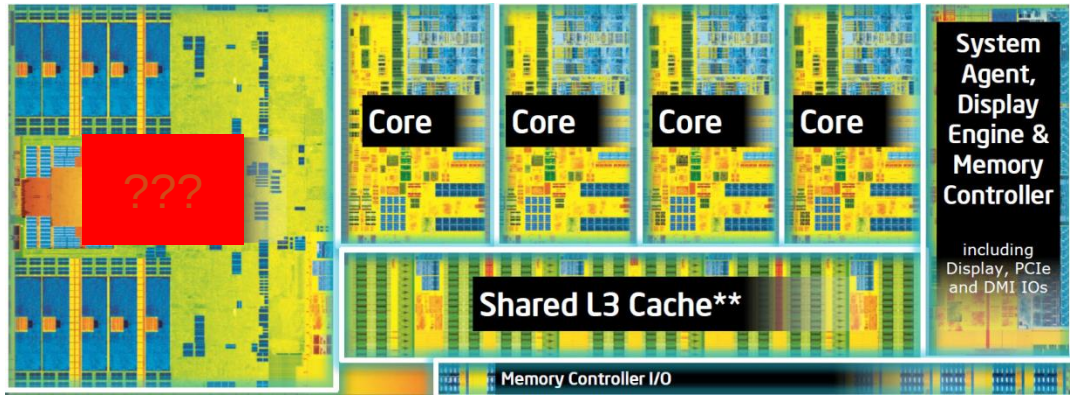


Explicit parallelism

- Then to support **explicit data & thread-level parallelism**
  - Hardware provides parallel resources, software specifies usage
  - Why? diminishing returns on instruction-level-parallelism
- First using (sub-word) vector instructions …
  - E.g., in Intel's SSE, one instruction does four parallel multiplies
- … adding support for multi-threaded programs …
  - Coherent caches, hardware synchronization primitives
- New architectures, e.g., programmable GPUs
  - Some attempts at convergence between CPUs and GPUs (e.g., Intel's Xeon Phi)

Multicore

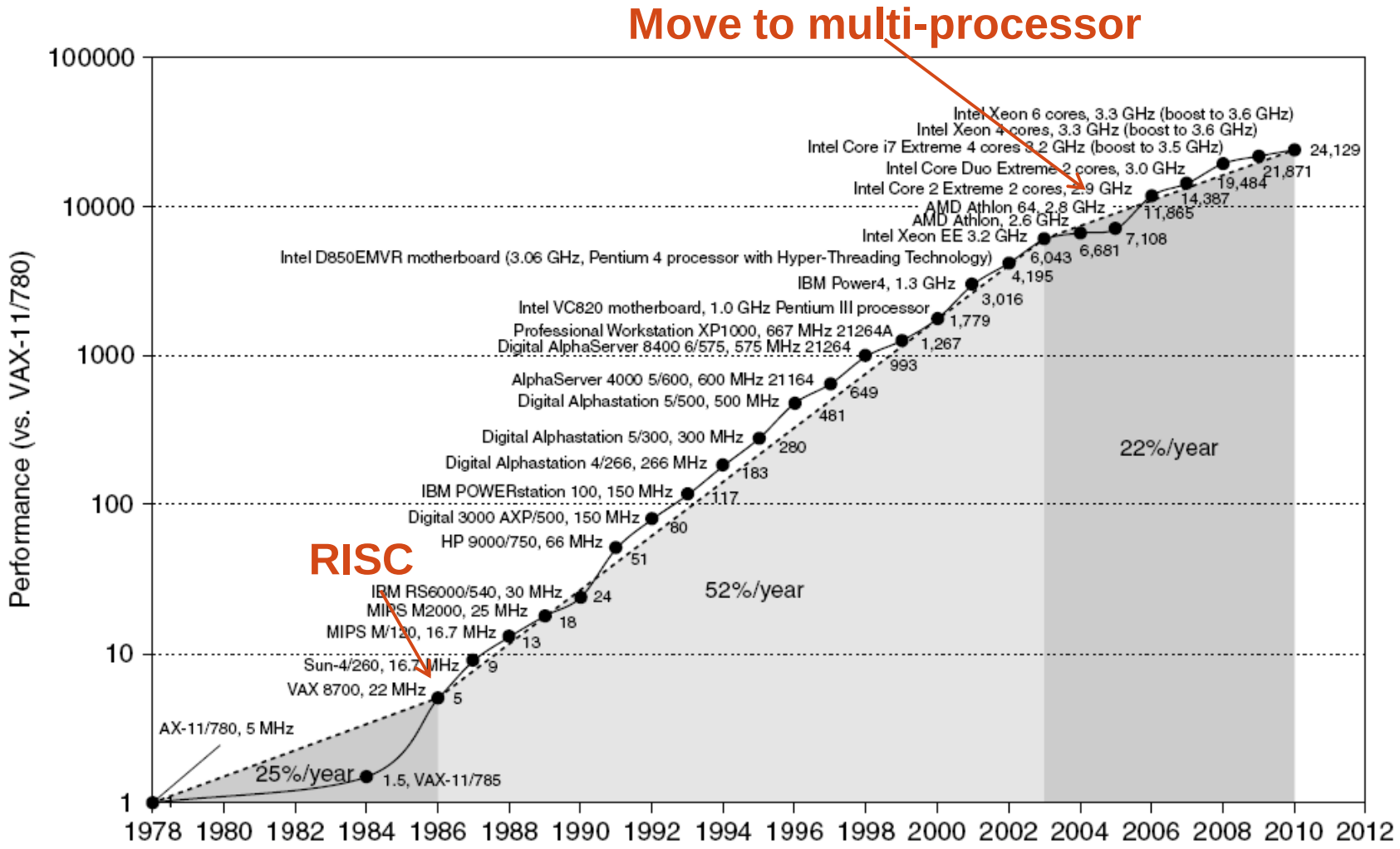
- Intel Core i7 (2013)
  - Application: desktop/server
  - Technology: 22nm *(25% of P4)*



  - 1.4B transistors *(30x)*
  - 177 mm^2 *(2x)*
  - 3.5 GHz to 3.9 Ghz *(~1x)*
  - 1.8 Volts *(~1x)*

  - 256-bit data *(2x)*
  - 14-stage pipelined datapath *(0.5x)*
  - 4 instructions per cycle *(~1x)*
  - Three levels of on-chip cache *(1.5x)*
  - Data-parallel “vector” (SIMD) instructions, hyperthreading
  - **Four-core multicore** (4x)

Performance over the years



Specialization

Hard to get parallel speedup for many applications

- Writing parallel software is hard!
- Extra cores give little benefit
- ➔ Can we put those transistors to better use?

Specialized processors are much more efficient

- Customized datapath for common operations (many instructions ➔ 1 cycle)
- Customized memories keep data near where its used
- Eliminate features that aren't needed (less power)

…But only worthwhile for the most important workloads

- Design & verification is expensive
- Software now must support custom hardware
- Wastes chip real estate when idle

System-on-chip

Qualcomm Snapdragon 835

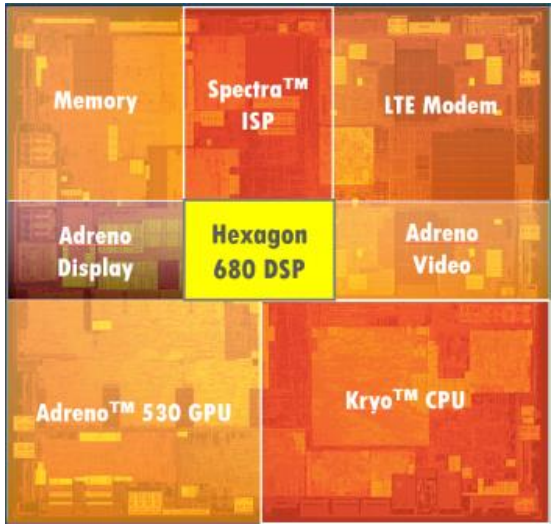
- Application: Mobile
- Technology: 10nm

- ARM CPUs – heterogeneous “big.LITTLE” design
  - 4 “performance” cores – 2.45 GHz, 2MB L2 cache
  - 4 “efficiency” cores – 1.9 GHz, 1MB L2 cache
  - “Performance” cores are 20% faster; “efficiency” cores used 80% of the time

- Graphics processing unit (GPU)
  - ~650 MHz
  - Specialized floating-point datapath, e.g., for interpolation of textures
  - Data-parallel: 16 pixels / clock
  - Processor dynamically finds & schedules work (“warp scheduling”)

- Digital signal processor (DSP)
  - Data-parallel SIMD architecture with 4 instructions / cycle
  - No floating-point
  - Compiler statically schedules parallelism (“VLIW”)

- Other custom accelerators (camera, modem, etc)



*Snapdragon 820
(only die shot I could find)

Architectures today

Multicore CPUs (e.g., Intel Xeon)

- Traditional hard-to-parallelize code – web serving
- Renewed focus on CPU microarchitecture – sequential performance still matters!

GPUs (e.g., Nvidia)

- “Embarrassingly parallel” code – science, graphics, DNNs
- Increasing programmability, converging towards traditional vector design

System-on-chip & domain-specialized accelerators

- Energy-efficiency – embedded, mobile, (datacenter – Google’s TPU???)
- Lots of open questions ...
  - How many accelerators do we need?
  - Which ones?
  - How specialized should they be?

Computer Architecture in Broad Strokes

What computer architects do

Given constraints of

- Technology
- Application

Use essential themes

- Locality (e.g., caching)
- Prediction / speculation
- Pipelining
- Parallelism
- Virtualization / indirection
- Specialization

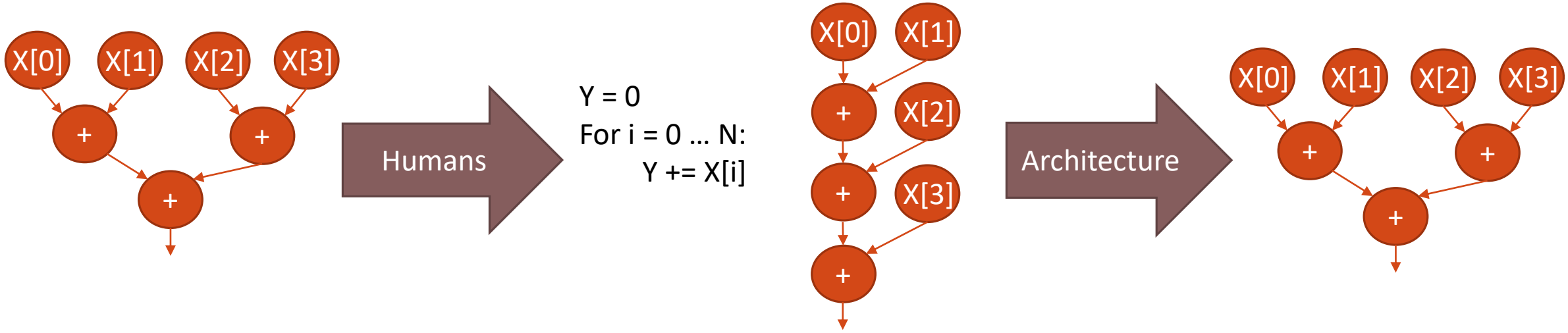
And, always, using abstraction…

Broad strokes: Parallelism

Computing is **fundamentally parallel** (this is naturally captured in hardware designs)

Parallelism is hard, so humans prefer **step-by-step recipes**

➔ Thus arose the **imperative computing model**

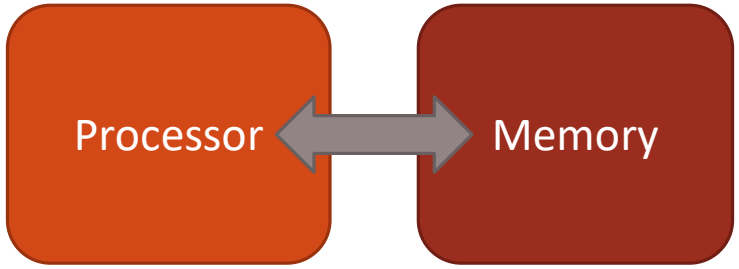


Unfortunately, doing things step-by-step is **abysmally slow and inefficient**

A recurring challenge is **recovering parallelism without over-burdening programmers**

Broad strokes: Processing vs. Memory

Computer scientists make a fundamental distinction between **processing** and **memory**



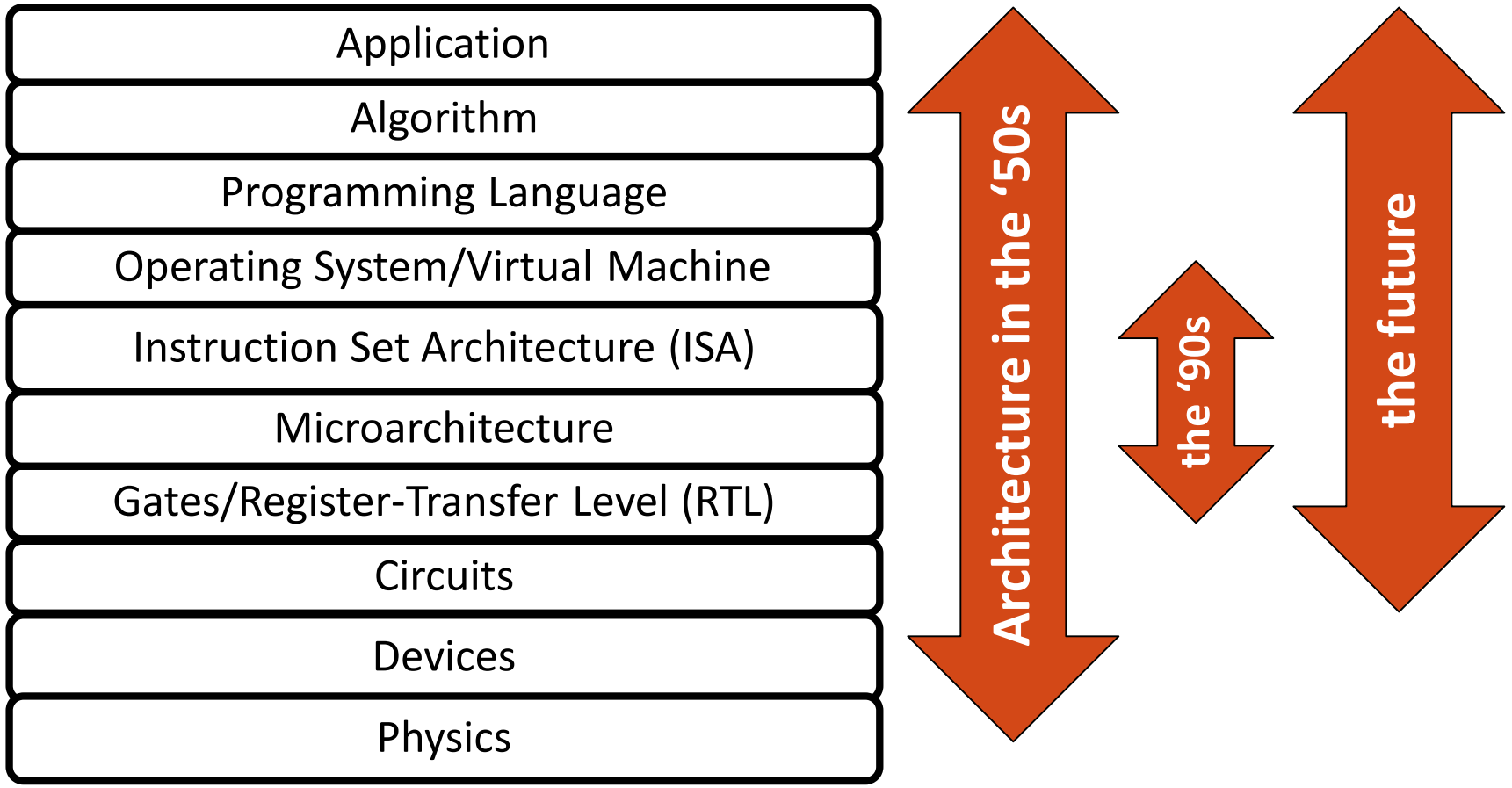
This makes sense, but it is a **choice** (contrast with, e.g., neural networks)

Historically, **computer science focuses on processing** (also true of architects)

But increasingly, **memory/communication is the primary challenge**

- Increasing data sizes + technology ➔ data is increasingly expensive
- Consistency of parallel updates to data
- Compute is easier to specialize
- Some recent designs attempt to eliminate this dichotomy (“processing in memory”)

Abstraction layers in modern systems



Broad strokes: Erosion of Familiar Abstractions

During mid ’80s-early ‘00s, processors got steadily faster each year

➔ Most computer scientists learned it was **safe to ignore computer architecture**

SURPRISE!

Technology scaling ends: first Dennard in early ‘00s, now Moore in next ~five years

Architectural limits: pipelining + ILP give diminishing returns (more on this later)

➔ Software must change

2000-2010: Multicore, GPGPU

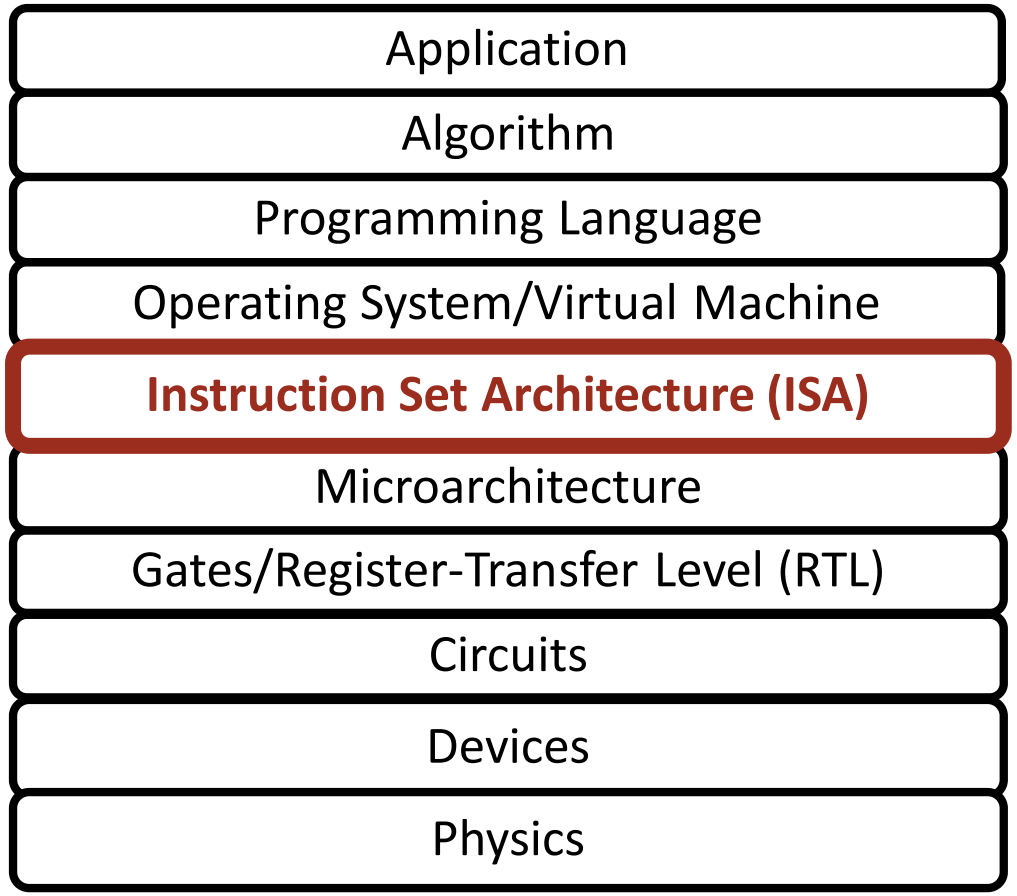
2010-now: Accelerators

Software folks must pay more attention to hardware! (Even if you don’t want to… CPUs have stopped scaling, but application’s haven’t.)

Instruction set architecture (ISA)

(MORE CONTENT IS AVAILABLE ONLINE IF USEFUL FOR ASSIGNMENTS…)

Abstraction layers in modern systems



ISA defines **functional contract** between hardware and software

I.e., what the hardware does and (doesn't) guarantee will happen

Abstraction & your program

High-level language

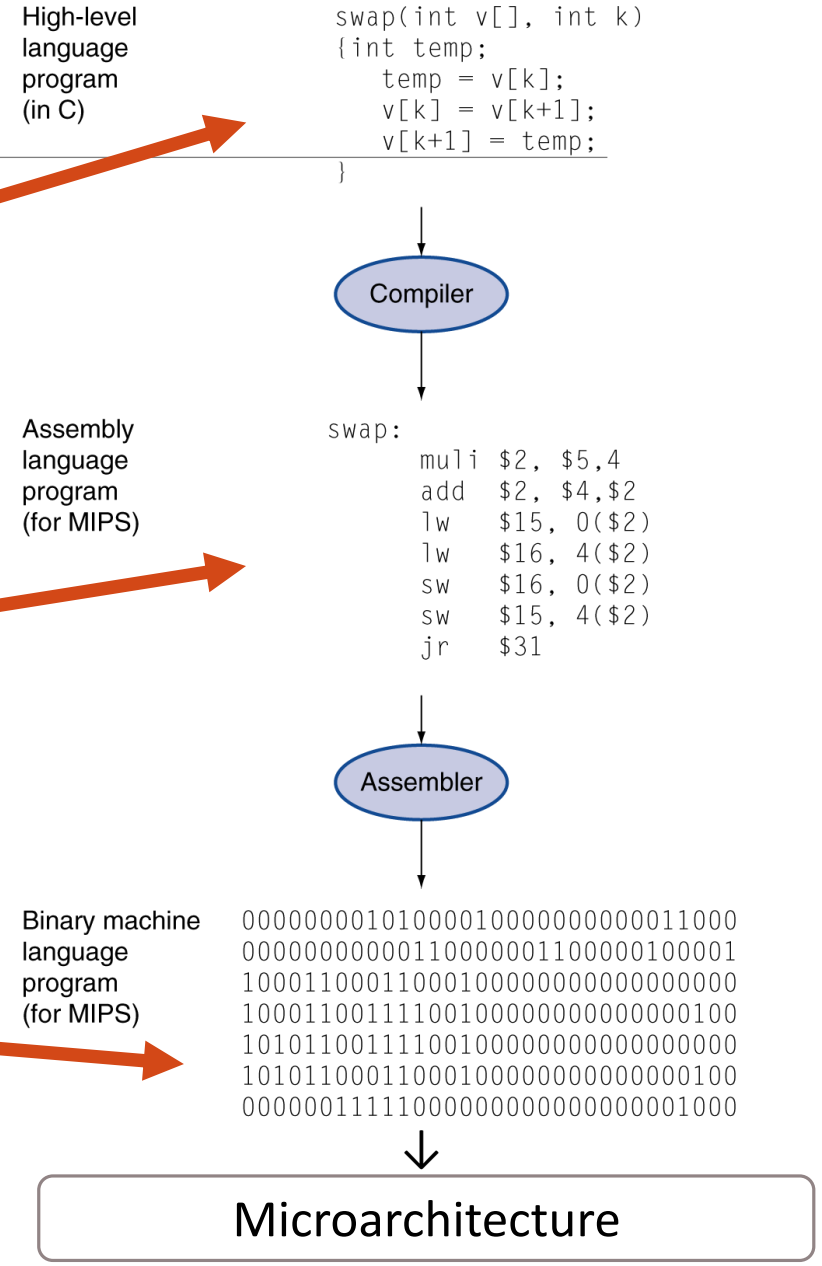
- Level of abstraction closer to problem domain
- Provides for productivity and portability

Assembly language

- Textual representation of instructions (ISA)

Hardware representation

- Binary representation of instructions (ISA)



Instruction set architecture (ISA)

The ISA defines the ***functional*** contract between the software and the hardware

The ISA is an ***abstraction*** that hides details of the implementation from the software

➔ The ISA is functional abstraction of the processor (a “mental model”)

- What operations can be performed
- How to name storage locations
- The format (bit pattern) of the instructions

ISA typically does **NOT** define

- Timing of the operations
- Power used by operations
- How operations/storage are implemented

If timing leaks information, is it Intel’s fault?
No, the abstraction is broken.

What goes into an ISA?

Operands

- How many?
- What kind?
- Addressing mechanisms

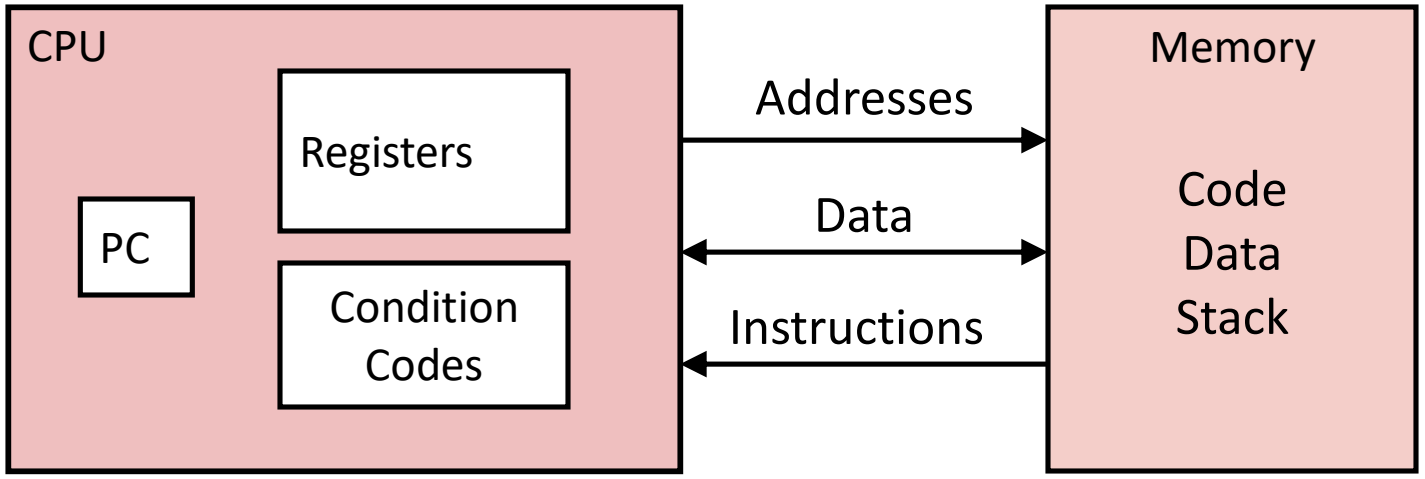
Operations

- What kind?
- How many?

Format/encoding

- Length(s) of bit pattern
- Which bits mean what

Assembly programmer's view



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "EIP" (IA32) or "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Operands ⇔ Machine model

Three basic types of machine

- Stack
- Accumulator
- Register

Two types of register machines

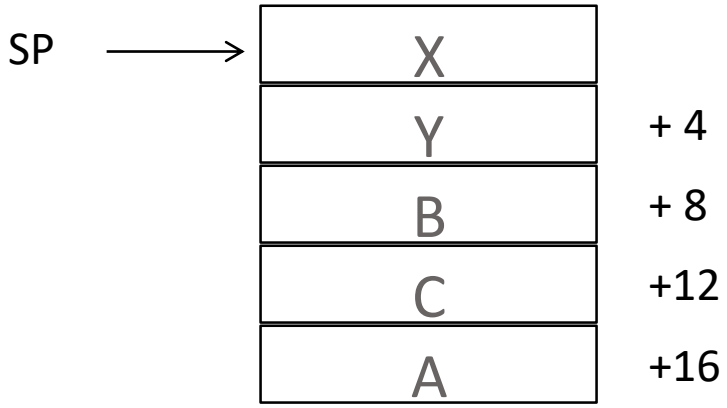
- Register-memory
  - Most operands in most instructions can be either a register or a memory address
- Load-store
  - Instructions are either load/store or register-based

Operands per instruction

Stack		
0 address	add	push(pop() + pop())
Accumulator		
1 address	add A	Acc ← Acc + mem[A]
Register-Memory		
2 address	add R1, A	R1 ← R1 + mem[A]
3 address	add R1, R2, A	R1 ← R2 + mem[A]
Load-Store		
3 address	add R1, R2, R3	R1 ← R2 + R3
2 address	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

Examples

Code for: A=X*Y – B*C

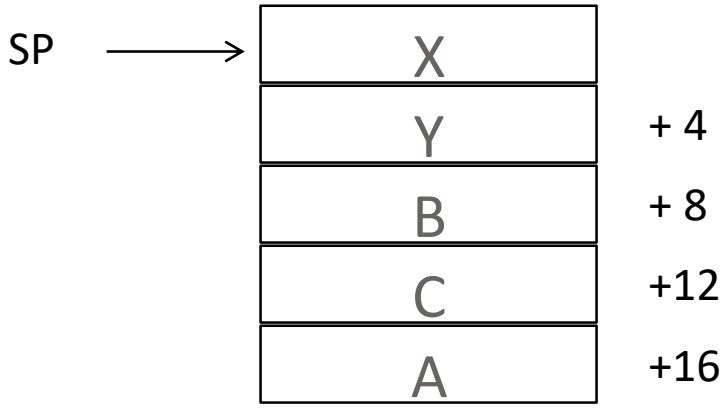


Stack

```
push    8(SP)
push    16(SP)
mult
push    4(sp)
push    12(sp)
mult
sub
st      20(sp)
pop
```

Examples

Code for: A=X*Y – B*C



Stack

```
push    8(SP)
push    16(SP)
mult
push    4(sp)
push    12(sp)
mult
sub
st      20(sp)
pop
```

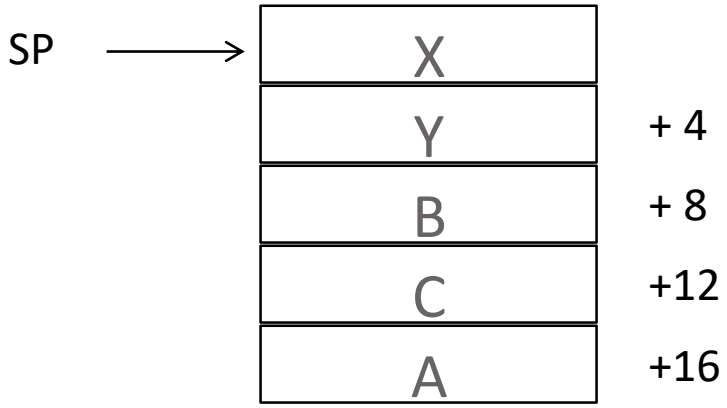
Accumulator

```
ld      8(SP)
mult    12(SP)
st      20(SP)
ld      4(SP)

mult    0(SP)
sub     20(sp)
st      16(sp)
```

Examples

Code for: A=X*Y – B*C



Stack

```
push    8(SP)
push    16(SP)
mult
push    4(sp)
push    12(sp)
mult
sub
st      20(sp)
pop
```

Accumulator

```
ld      8(SP)
mult    12(SP)
st      20(SP)
ld      4(SP)

mult    0(SP)
sub     20(sp)
st      16(sp)
```

reg-mem

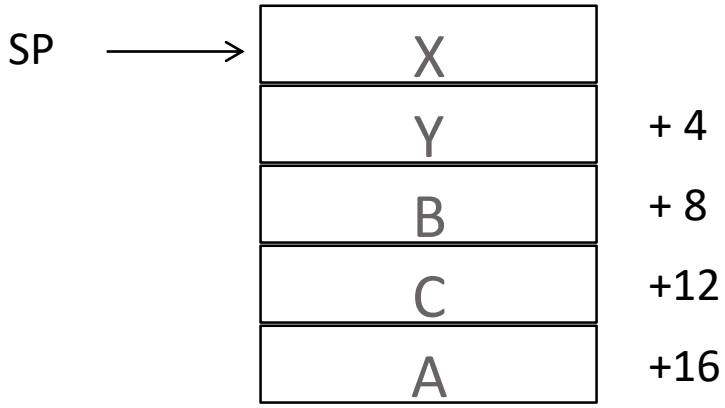
```
mult    R1,8(SP),12(SP)

mult    R2,0(SP),4(SP)

sub     16(sp),R2,R1
```

Examples

Code for: A=X*Y – B*C



Stack

```
push    8(SP)
push    16(SP)
mult
push    4(sp)
push    12(sp)
mult
sub
st      20(sp)
pop
```

Accumulator

```
ld      8(SP)
mult    12(SP)
st      20(SP)
ld      4(SP)

mult    0(SP)
sub     20(sp)
st      16(sp)
```

reg-mem

```
mult    R1,8(SP),12(SP)

mult    R2,0(SP),4(SP)

sub     16(sp),R2,R1
```

ld/st

```
ld      r1,8(SP)
ld      r2,12(SP)
ld      r3,4(SP)
ld      r4,0(SP)
mult    r5,r1,r2
mult    r6,r3,r4
sub     r7,r6,r5
st      16(SP),r7
```

Machine model tradeoffs

Stack and Accumulator:

- Each instruction encoding is short
- IC is high
- Very simple exposed architecture

Register-Memory:

- Instruction encoding is much longer
- More work per instruction
- IC is low
- Architectural state more complex

Load/Store:

- Medium encoding length
- Less work per instruction
- IC is high
- Architectural state more complex

ISA design goals

Ease of programming / Code generation	(software perspective)
Ease of implementation	(hardware perspective)
Good performance	
Compatibility	
Completeness (eg, Turing)	
Compactness – reduce program size	
Scalability / extensibility	
Features: Support for OS / parallelism / …	
Etc	

Ease of programming

The ISA should make it easy to express programs and make it easy to create efficient programs.

Who is creating the programs?

- Early Days: Humans. Why?

Ease of programming

The ISA should make it easy to express programs and make it easy to create efficient programs.

Who is creating the programs?

- Early Days: Humans.
  - No real compilers
  - Resources very limited
  - Q: What does that mean for the ISA designer?
  - A: High-level operations

- Modern days (~1980 and beyond): Compilers
  - Today's optimizing compiler do a much better job than most humans
  - Q: What does that mean for the ISA designer?
  - A: Fine-grained, low-level operations

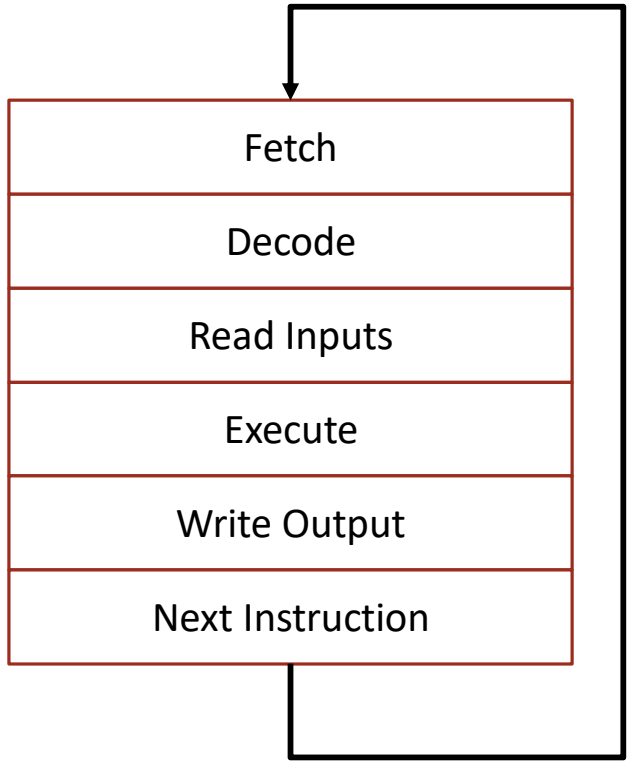
Ease of implementation

ISA shouldn't get in the way of optimizing implementation

Examples:

- Variable length instructions
- Varying instruction formats
- Implied registers
- Complex addressing modes
- Precise interrupts
- Appearance of atomic execution

Simple processor pipeline:



“The Iron Law of Performance”

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

What determines each factor? How does ISA impact each?

Instructions / program = **dynamic** instruction count (not code size)

- Determined by program, compiler, and ISA

Cycles / instruction (CPI)

- Determined by ISA, μarch, program, and compiler

Seconds / cycle (critical path)

- Determined by μarch and technology

Cycles per instruction (CPI)

Different instruction classes take different numbers of cycles

In fact, even the same instruction can take a different number of cycles

- Example?

When we say CPI, we really mean: ***Weighted CPI***

$$CPI = \frac{Clock\ cycles}{Instruction\ count} = \sum_{i=1}^{n} CPI_i \times \frac{Instruction\ count_i}{Instruction\ count}$$

How to improve performance

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

1. Reduce instruction count
2. Reduce cycles per instruction
3. Reduce cycle time

But there is a tension between these…

CPI example – which machine is faster?

Computer A: Cycle time = 250ps, CPI = 2.0

Computer B: Cycle time = 500ps, CPI = 1.2

Same ISA

$$CPU\ time_A$$
$$= Instruction\ count \times CPI_A \times Cycle\ time_A$$
$$= Instruction\ count \times 2 \times 250ps$$
$$= Instruction\ count \times 500ps$$

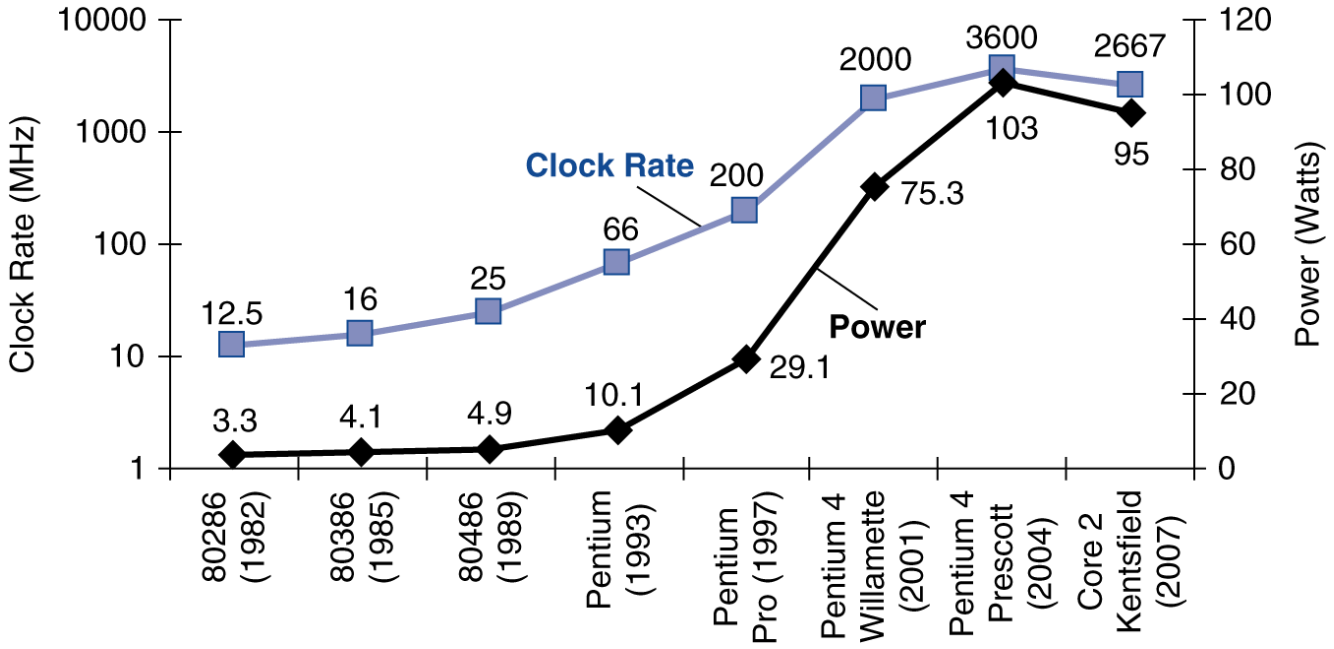
A is faster…

$$CPU\ time_b$$
$$= Instruction\ count \times CPI_B \times Cycle\ time_B$$
$$= Instruction\ count \times 1.2 \times 500ps$$
$$= Instruction\ count \times 600ps$$

$$\frac{CPU\ time_B}{CPU\ time_A} = \frac{600ps}{500ps} = 1.2$$

…by this much

History: CMOS & power



In CMOS IC technology:

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

Compatibility

"Between 1970 and 1985 many thought the primary job of the computer architect was the design of instruction sets. …The educated architect was expected to have strong opinions about the strengths and especially the weaknesses of the popular computers. **The importance of binary compatibility in quashing innovation** in instruction set design was unappreciated by many researchers and textbook writers, giving the impression that many architects would get a chance to design an instruction set." - H&P, Appendix A

Compatibility

ISA separates interface from implementation

Thus, many different implementations possible

- IBM/360 first to do this and introduce 7 different machines all with same ISA
- Intel from 8086 → core i7 → Xeon Phi → ?
- ARM ISA mobile → server

Protects software investment

Important to decide what should be exposed and what should be kept hidden.

- E.g., MIPS “branch delay slots”

RISC vs CISC

RISC vs CISC

RISC: Reduced Instruction Set Computer

- Introduced Early 80's
- RISC-I (Berkeley), MIPS (Stanford), IBM 801
- Today: ARM (…kinda), RISC-V (Berkeley)

CISC: Complex Instruction Set Computer

- What everything was before RISC
- VAX, x86, 68000
- Today: x86

Outcome:

- RISC in academy
- CISC in high-performance processors, but…
- RISC in embedded + under the covers

Basic comparison

CISC

- variable length instructions: 1-321 bytes
- GP registers+special purpose registers+PC+SP+conditions
- Data: bytes to strings
- memory-memory instructions
- special instructions: e.g., crc, polyf, …

RISC

- fixed length instructions: 4 bytes
- GP registers + PC
- load/store with few addressing modes

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8*, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 /r	ADD *r/m8*, r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /r	ADD *r8*, r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 /r	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /r	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

How did RISC happen?

Pre-1980

- Lots of hand written assembly
- Compiler technology in its infancy
- Multi-chip implementations
- Small memories at ~CPU speed

Early 80's

- VLSI makes single chip processor possible
  (But only if very simple)
- Compiler technology improving

How did RISC happen?

Pre-1980

- Lots of hand written assembly
- Compiler technology in its infancy
- Multi-chip implementations
- Small memories at ~CPU speed

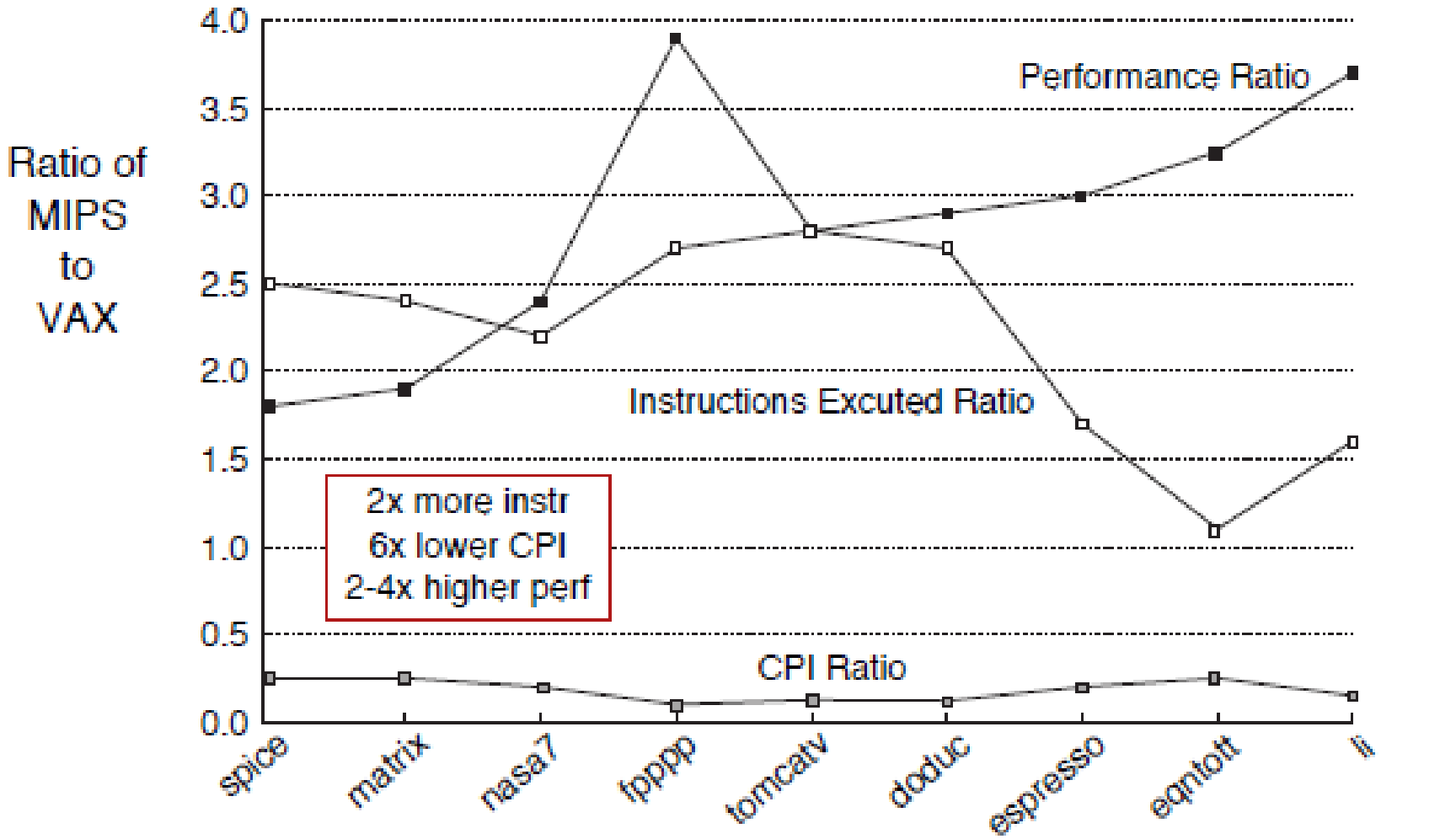
Early 80's

- VLSI makes single chip processor possible
  (But only if very simple)
- Compiler technology improving

RISC goals:

- **Enable single-chip CPU**
- **Rely on compiler**
- **Aim for high frequency & low CPI**

MIPS v. VAX



-- H&P, Appendix J, from Bhandarkar and Clark, 1991

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

Schools of ISA design & performance

$$CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

Complex instruction set computer (CISC)

- Complex instructions ➔ lots of work per instruction ➔ fewer instructions per program
- But… more cycles per instruction & longer clock period
- *Modern µarch gets around most of this!*

Reduced instruction set computer (RISC)

- Fine-grain instructions ➔ less work per instruction ➔ more instructions per program
- But… lower cycles per instruction & shorter clock period
- *Heavy reliance on compiler to "do the right thing"*

The case for RISC

CISC is fundamentally handicapped

At any given technology, RISC implementation will be 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 ...

➔ RISC will always be one step ahead of CISC!

What actually happened?

Pre-1980

- lots of hand written assembly
- Compiler technology in its infancy
- multi-chip implementations
- Small memories at ~CPU speed

Early 80's

- VLSI makes single chip processor possible
  (But only if very simple)
- Compiler technology improving

Late 90's

- CPU speed vastly faster than memory speed
- More transistors makes μops possible

CISC's 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 internally **(μops)**

```
push $eax →      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!*
  - "OoO" also helps x86 more (because ISA limits compiler)
- Different **μops** for different designs
  - **Not part of the ISA specification → Implementation flexibility**

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)

More About Micro-ops

Two forms of μops "cracking"

- Hard-coded logic: fast, but complex (for insn in few μops)
- Table: slow, but "off to the side", doesn't complicate rest of machine
  - Handles the really complicated instructions

Core precept of architecture:

Make the common case fast, make the rare case correct.

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: μISA, many other tricks
- What about power efficiency?
  - Somewhat…RISC is most power-efficient today

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, cloud services
- **Open question: will “ARM compatibility” be the next x86?**

Course logistics

Logistics

Lectures

Paper readings, discussions & reviews

Labs

Project

Exams

Lectures

Please come

Participate! (10% of grade)

Lecture schedule and slides are online

- Until Exam 1: Memory hierarchy & parallelism
- After Exam 1: Microarchitecture & recent research topics

Labs

2 labs early in semester

- 5% grade each

Work in groups of 2-3

Goal:

- Become familiar with some tools
- Understand performance measurement
- Understand optimization aka
  How architecture affects use

First lab out this week! (more info next time)

Paper readings, discussion & reviews

We are trying something new ➔ feedback is welcome + we'll respond to it!

Each lecture has an associated paper

Reviews

- Before each discussion, pick two papers from intervening lectures to review
- Max half-page summary + discussion of the paper
  - What's the main idea? What problem is it solving? How does it solve it? How does it evaluate the solution?
- Include **3 questions you would ask the authors**
- 10% of grade – more importantly, an essential skill

Discussion

- You will each present once per semester (~20 total)
- Instructor (that's me) will then lead an open discussion of the paper

Project

Main focus, takes ~half of semester!

- 40% of grade

Do some “real” research

Work in groups of 2-3

Timeline:

Propose a topic	9/30
Meet w/ instructor	9/30 – 10/11
Start working	10/11
Milestone presentations	11/1 – 11/6
Poster	12/6
Final report	12/8

Exams

2 in-class exams

- 90 mins each
- 15% grade each

Closed book

Not cumulative

Exam 1: 10/4

Exam 2: 12/2

Let us know **now** if you cannot make these dates

Overview

Architecture: physics → applications

Constantly changing field:

- New problems
- New solutions
- …But many common patterns and useful insights

One must understand architecture to understand computer systems

What would you do with 1 trillion transistors?