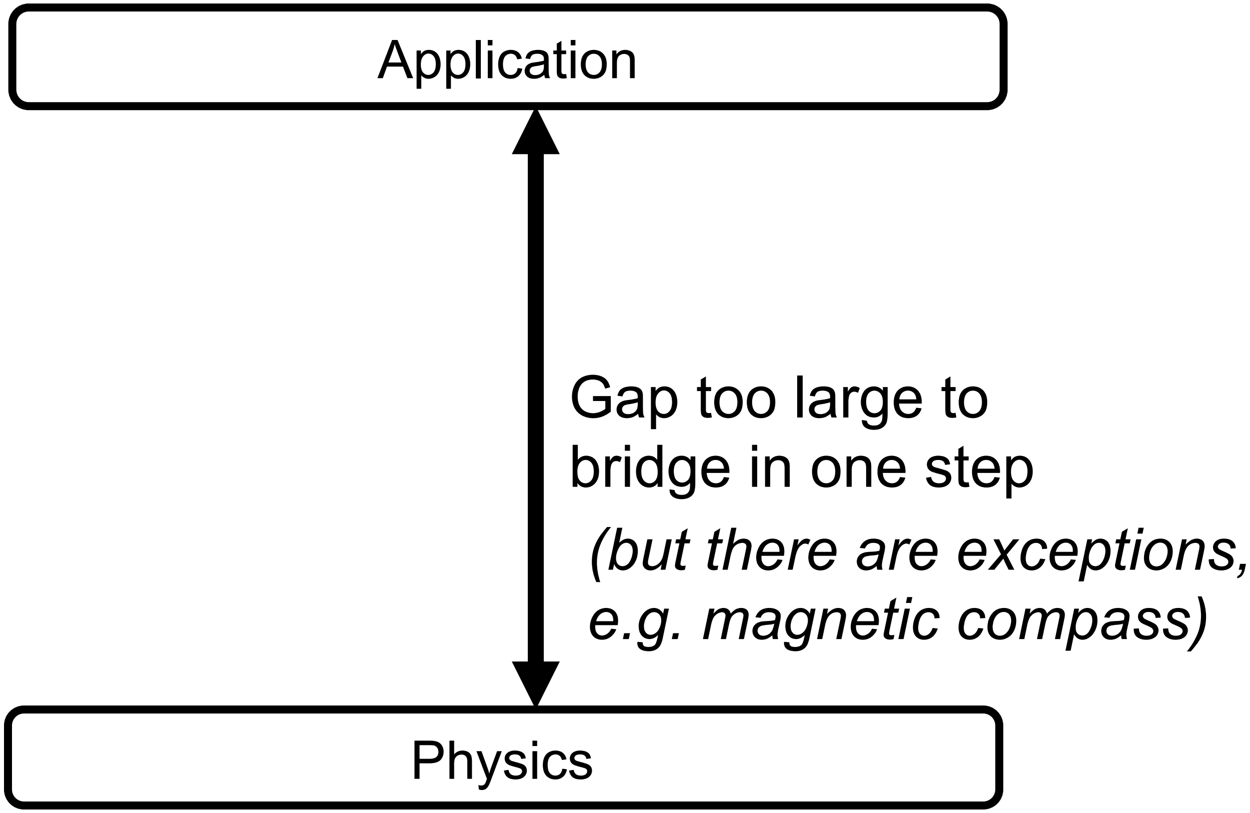
CS 152 Computer Architecture and Engineering

Lecture 1 - Introduction

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

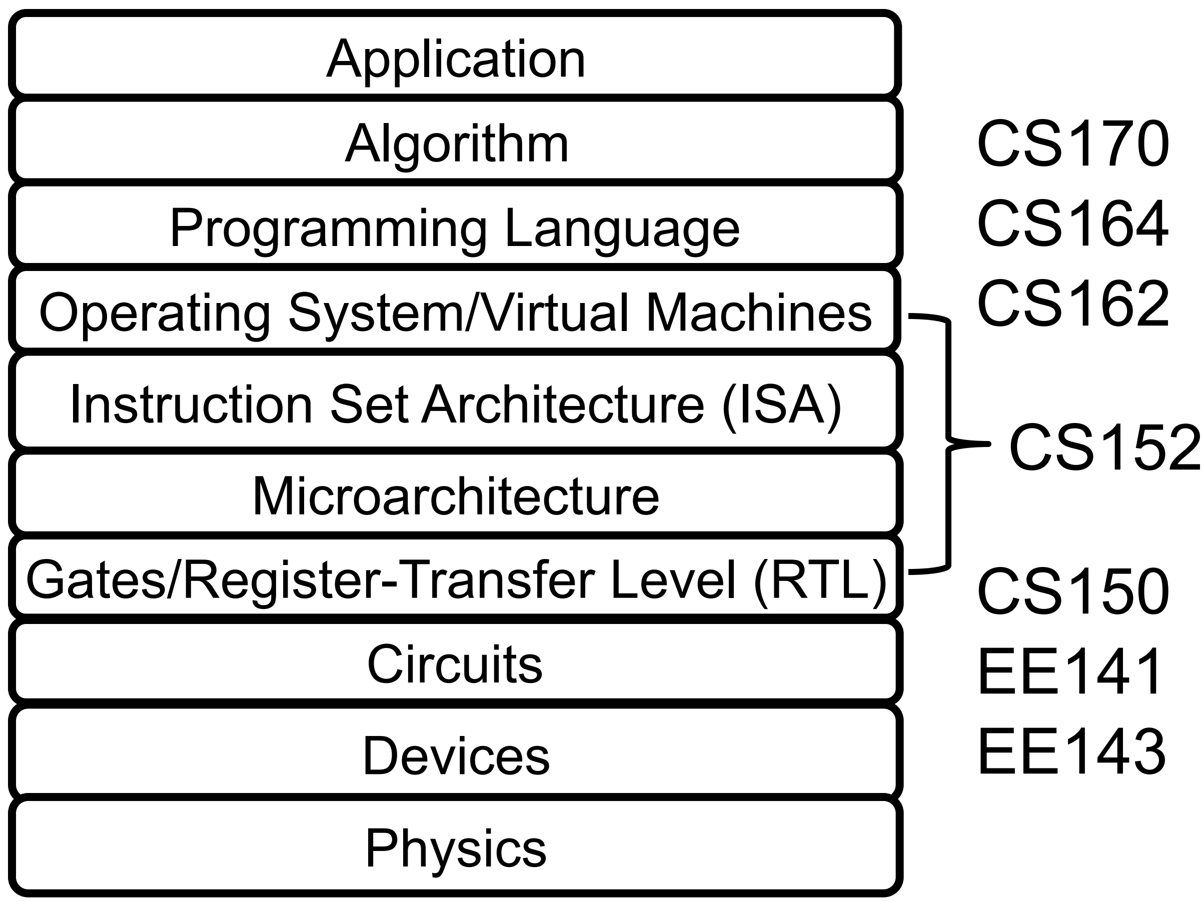
http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

What is Computer Architecture?



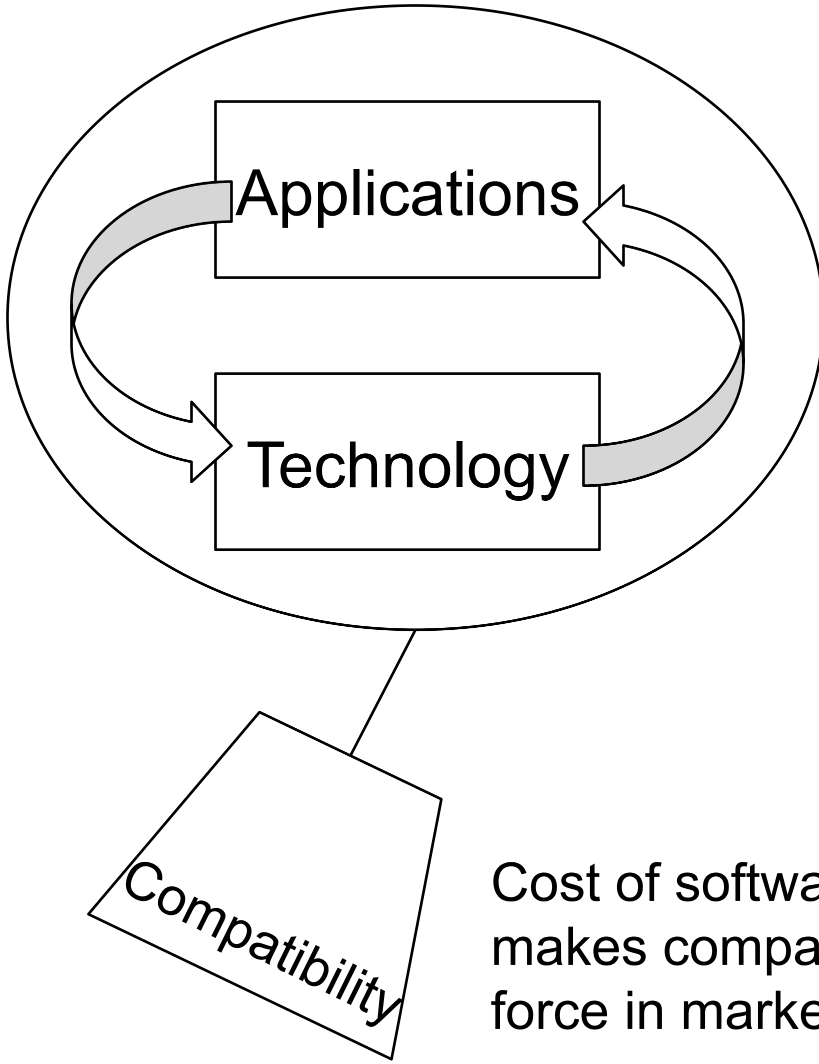
In its broadest definition, computer architecture is the *design of the abstraction layers* that allow us to implement information processing applications efficiently using available manufacturing technologies.

Abstraction Layers in Modern Systems



Architecture continually changing

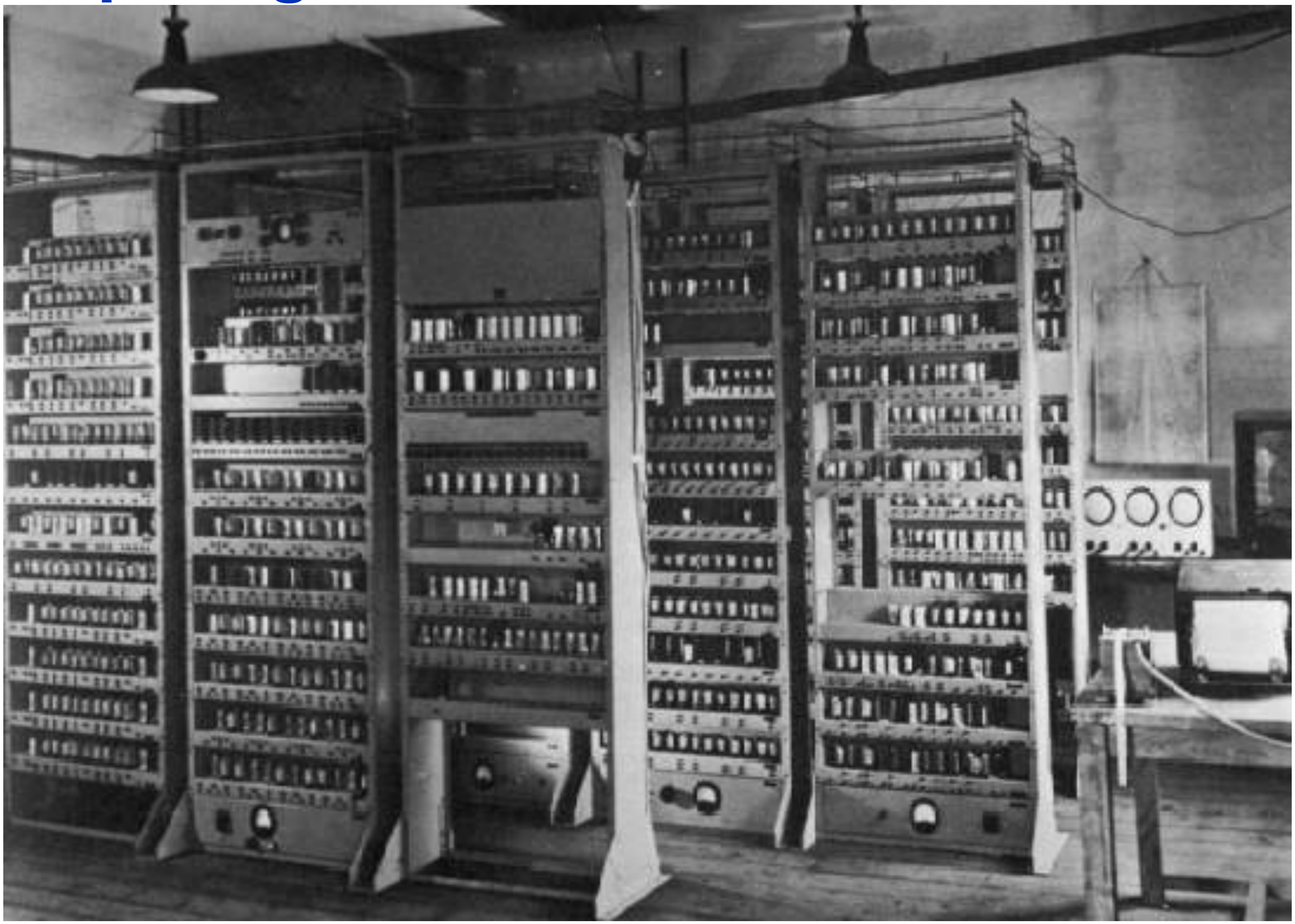
Applications suggest how to improve technology, provide revenue to fund development



Improved technologies make new applications possible

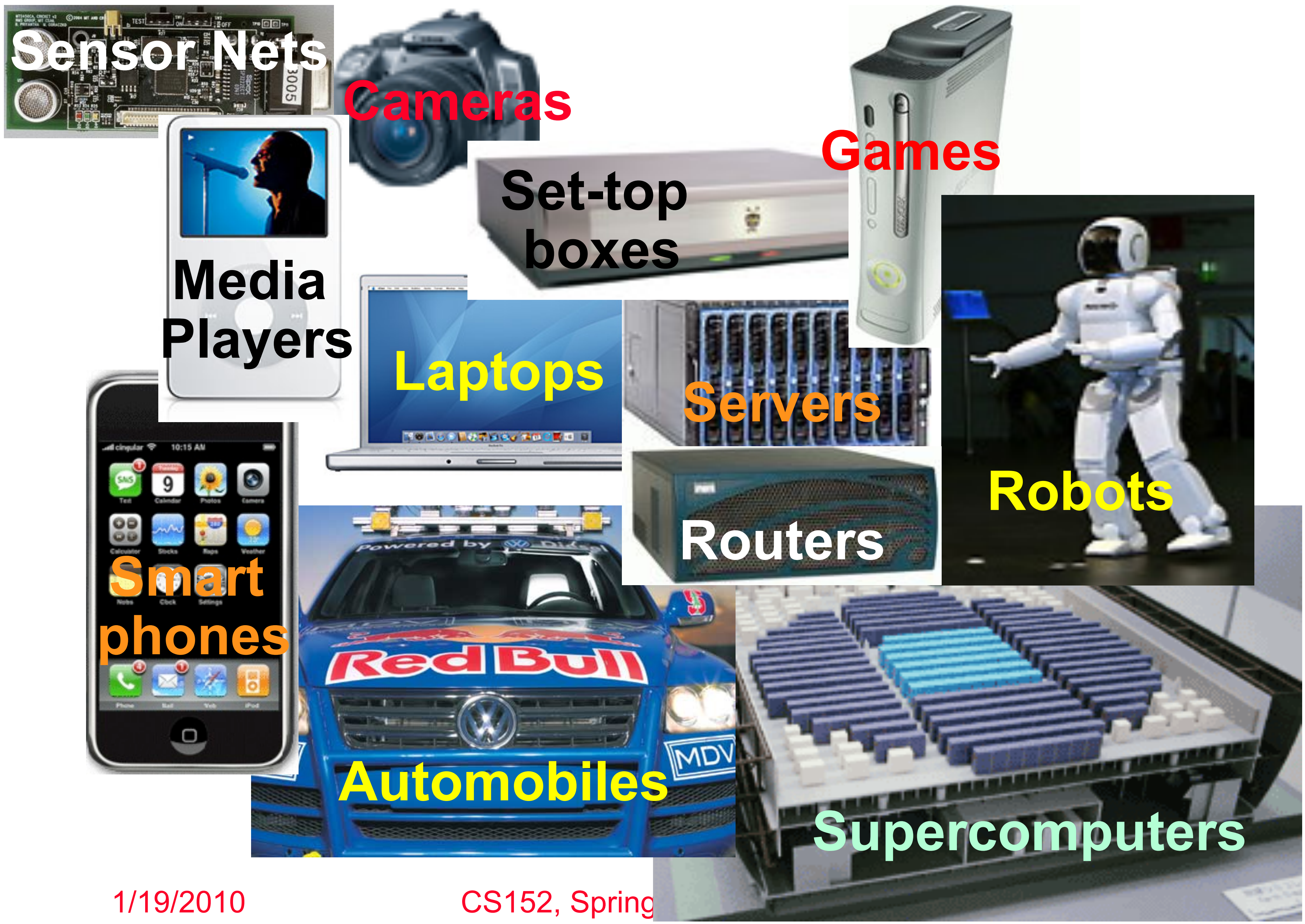
Cost of software development makes compatibility a major force in market

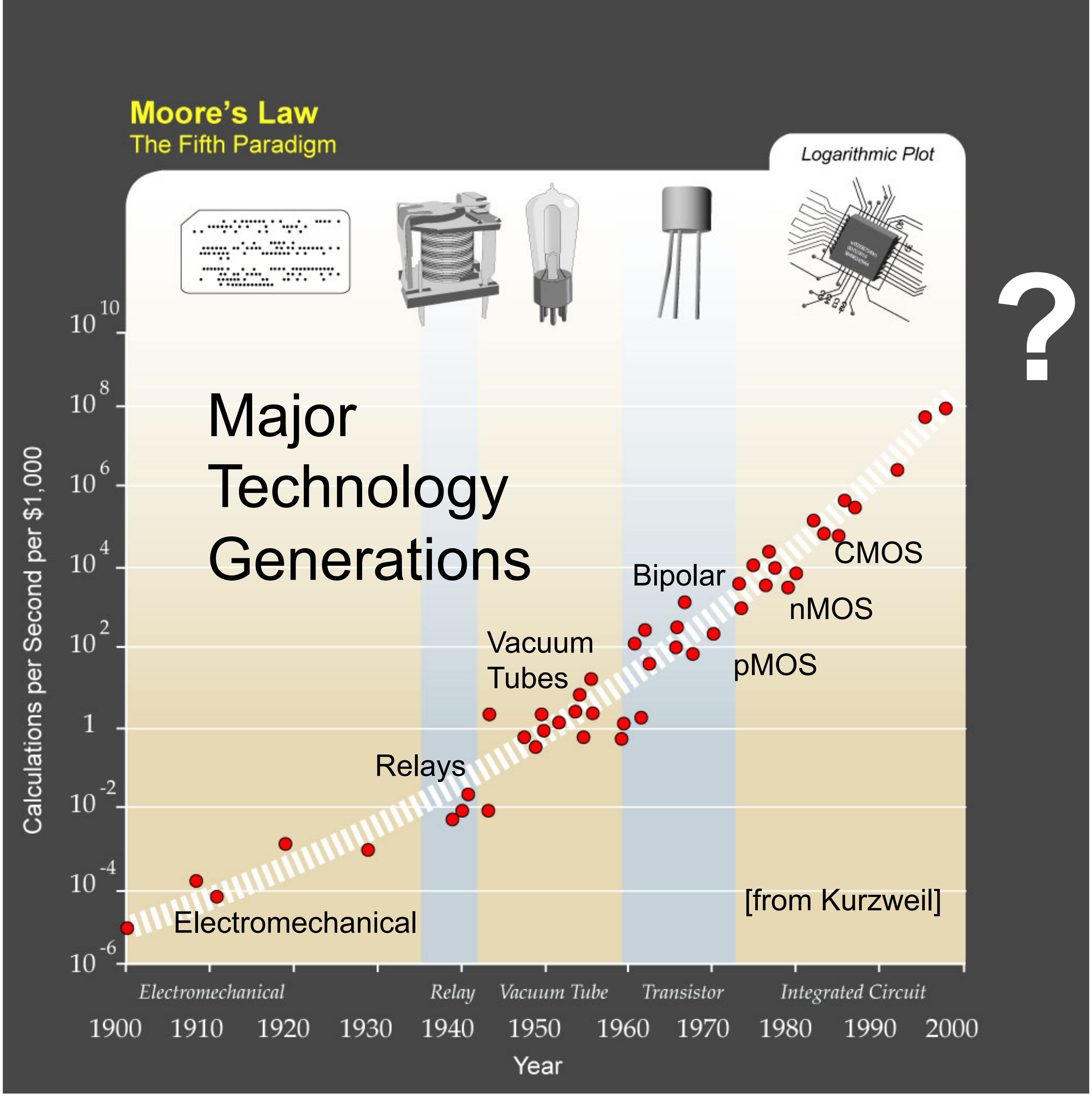
Computing Devices Then…



EDSAC, University of Cambridge, UK, 1949

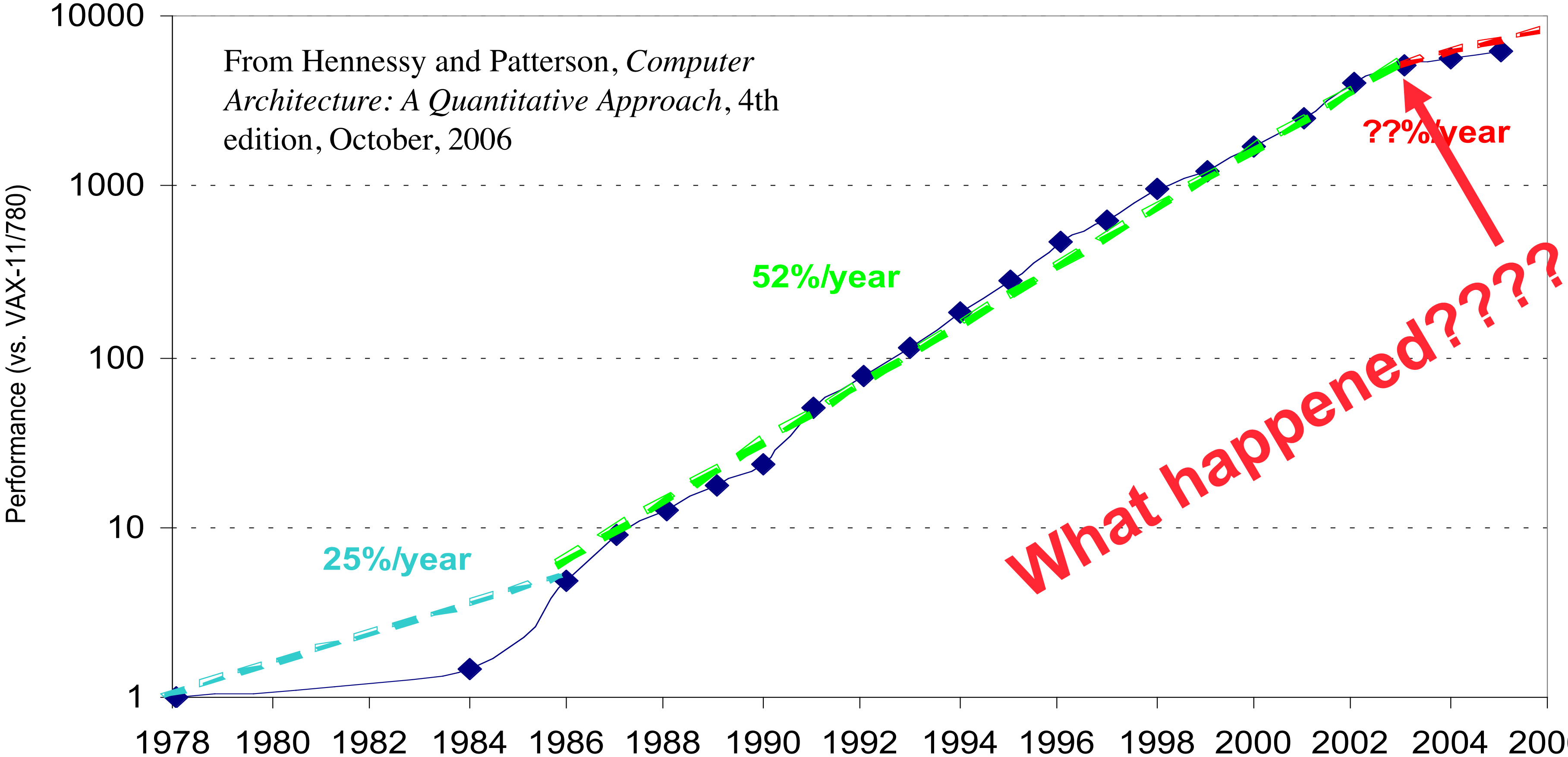
Computing Devices Now





Moore's Law
The Fifth Paradigm
Logarithmic Plot
Major Technology Generations
Calculations per Second per $1,000
10^10
10^8
10^6
10^4
10^2
1
10^-2
10^-4
10^-6
Electromechanical
Relays
Vacuum Tubes
Bipolar
pMOS
nMOS
CMOS
[from Kurzweil]
Electromechanical
Relay
Vacuum Tube
Transistor
Integrated Circuit
1900
1910
1920
1930
1940
1950
1960
1970
1980
1990
2000
Year
?

Uniprocessor Performance



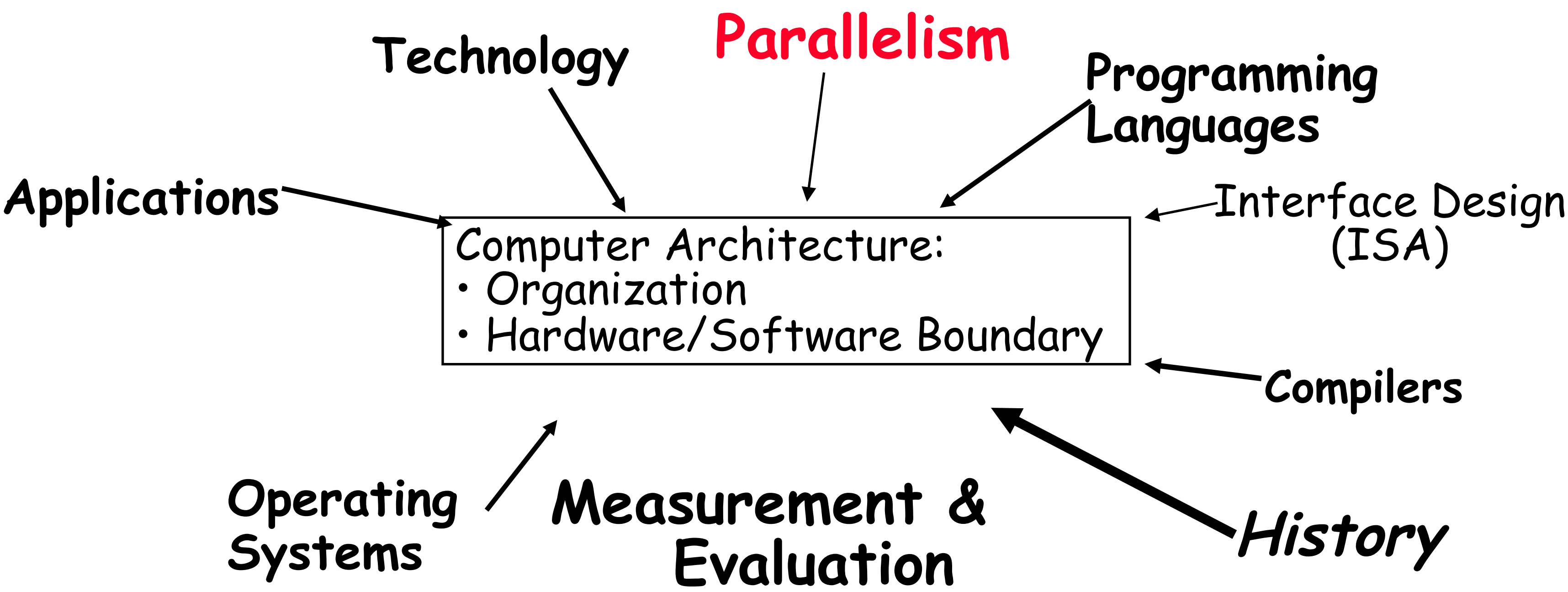
- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

The End of the Uniprocessor Era

Single biggest change in the history of computing systems

CS 152 Course Focus

Understanding the design techniques, machine structures, technology factors, evaluation methods that will determine the form of computers in 21st Century



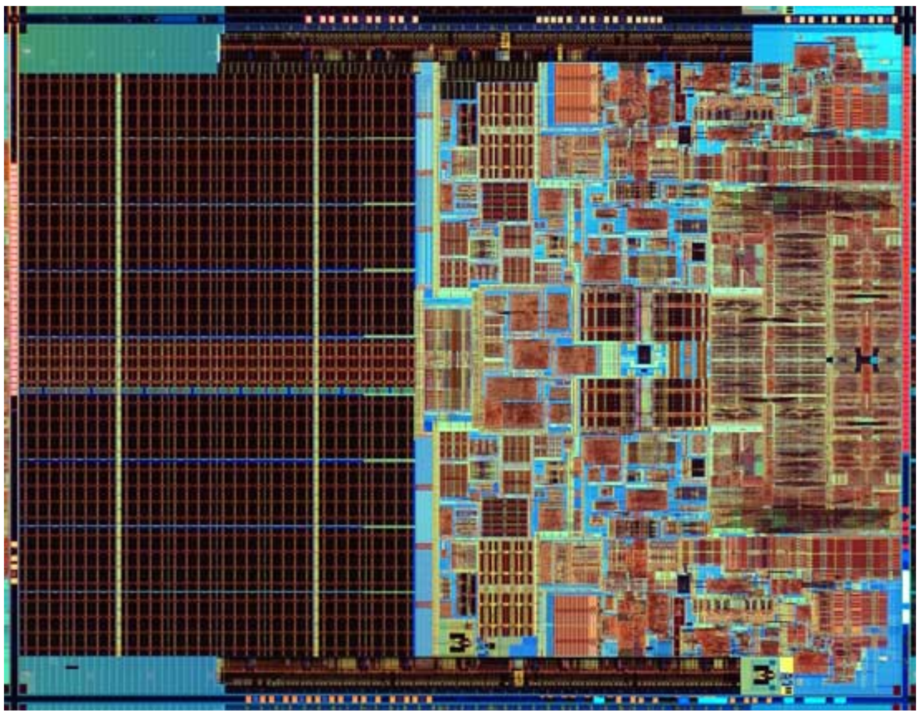
The "New" CS152

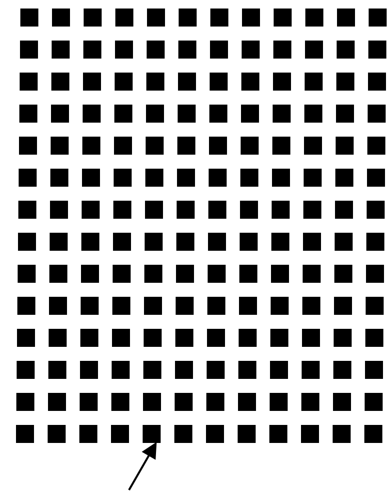
- New CS152 focuses on interaction of software and hardware
  – more architecture and less digital engineering.
- No FPGA design component
  – Take CS150 for digital engineering with FPGAs
  – Or CS250 for digital VLSI design with ASIC technology
- Much of the material you'll learn this term was previously in CS252
  – Some of the current CS61C, I first saw in CS252 nearly 20 years ago!
  – Maybe every 10 years, shift CS252->CS152->CS61C?
- Class contains labs based on various different machine designs
  – Experiment with how architectural mechanisms work in practice on real software.

The "New" CS152 Executive Summary

The processor your predecessors built in CS152

What you'll understand and experiment with in the new CS152





Plus, the technology behind chip-scale multiprocessors (CMPs)

CS152 Administrivia

Instructor: Prof. Krste Asanovic

Office: 579 Soda Hall, **krste@eecs**

Office Hours: Mon. 1:30-2:30PM (email to confirm), 579 Soda

T. A.: Andrew Waterman, **waterman@eecs**

Office Hours: TBD

Lectures: Tu/Th, 9:30-11AM, 310 Soda

Section: Th 2PM, 320 Soda

Text: *Computer Architecture: A Quantitative Approach, 4th Edition* (Oct, 2006)

Readings assigned from this edition, don't use earlier Eds.

Web page: **http://inst.eecs.berkeley.edu/~cs152**

Lectures available online ~6AM before class

CS152 Structure and Syllabus

Five modules

1. Simple machine design (ISAs, microprogramming, unpipelined machines, Iron Law, simple pipelines)
2. Memory hierarchy (DRAM, caches, optimizations) plus virtual memory systems, exceptions, interrupts
3. Complex pipelining (score-boarding, out-of-order issue)
4. Explicitly parallel processors (vector machines, VLIW machines, multithreaded machines)
5. Multiprocessor architectures (cache coherence, memory models, synchronization)

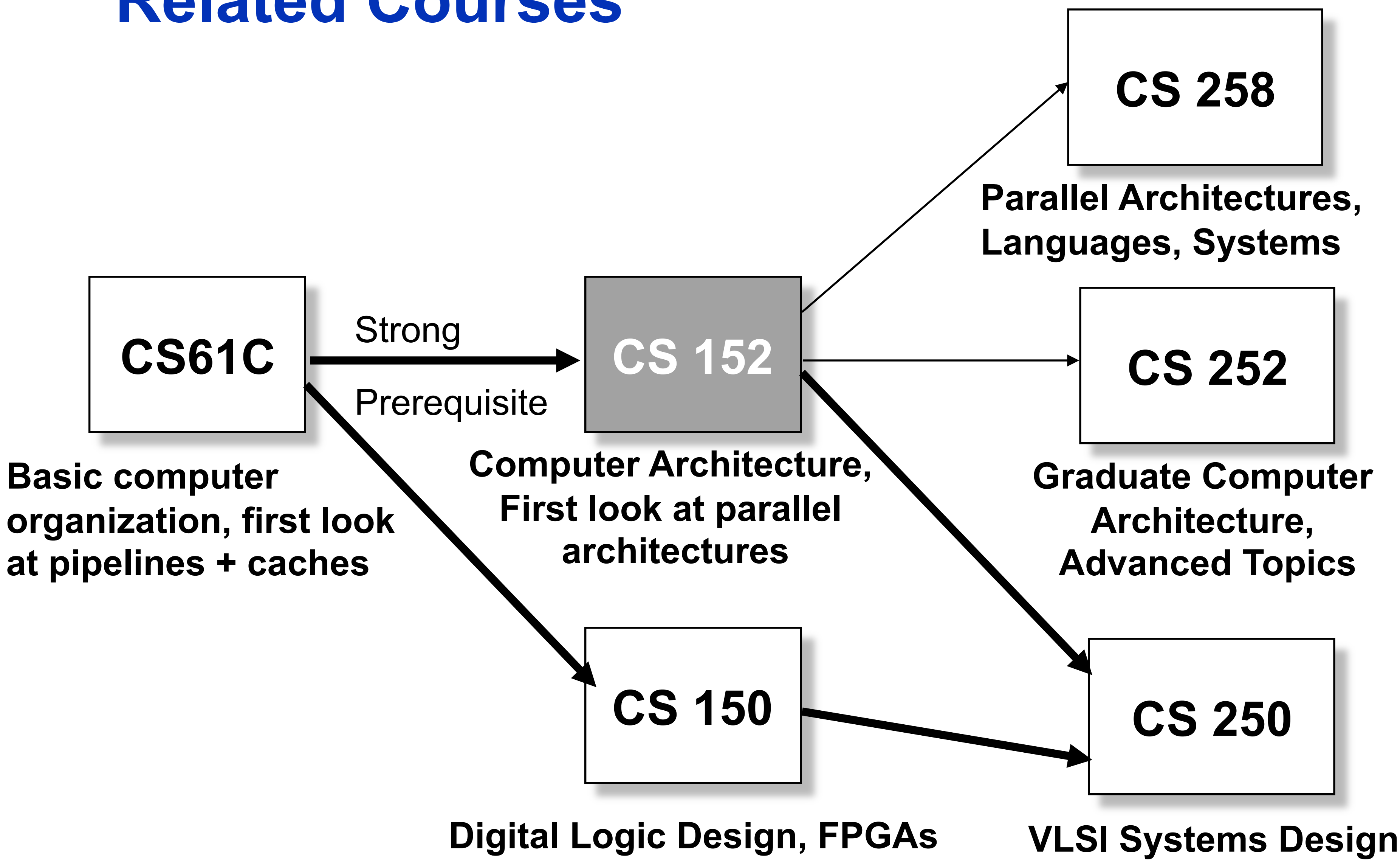
CS152 Course Components

- 20% Problem Sets (one per module)
  - Intended to help you learn the material. Feel free to discuss with other students and instructors, but must turn in your own solutions. Grading based mostly on effort, but quizzes assume that you have worked through all problems. Solutions released after PSs handed in.
- 40% Quizzes (one per module)
  - In-class, closed-book, no calculators or computers.
  - Based on lectures, problem sets, and labs
- 40% Labs (one per module)
  - Labs use advanced full system simulators (Virtutech Simics)
  - Directed plus open-ended sections to each lab

CS152 Labs

- Each lab has directed plus open-ended assignments
  - Roughly 50/50 split of grade for each lab
- Directed portion is intended to ensure students learn main concepts behind lab
  - Each student must perform own lab and hand in their own lab report
- Open-ended assigment is to allow you to show your creativity
  - Roughly a one day "mini-project"
    - » E.g., try an architectural idea and measure potential, negative results OK (if explainable!)
  - Students can work individually or in groups of two or three
  - Group open-ended lab reports must be handed in separately
  - Students can work in different groups for different assignments
- Lab reports must be readable English summaries – not dumps of log files!!!!!!

Related Courses



Computer Architecture: A Little History

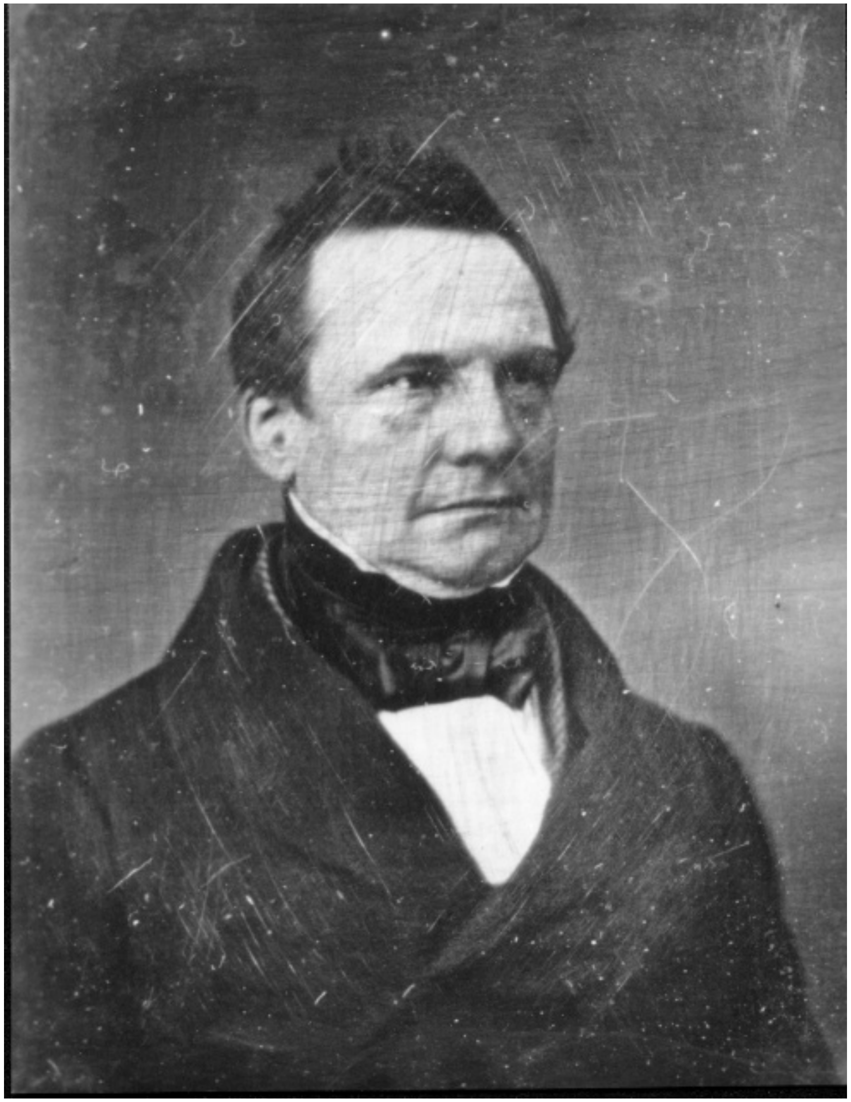
Throughout the course we'll use a historical narrative to help understand why certain ideas arose

Why worry about old ideas?

- Helps to illustrate the design process, and explains why certain decisions were taken
- Because future technologies might be as constrained as older ones
- Those who ignore history are doomed to repeat it
  - Every mistake made in mainframe design was also made in minicomputers, then microcomputers, where next?

Charles Babbage 1791-1871

Lucasian Professor of Mathematics, Cambridge University, 1827-1839



Charles Babbage

- *Difference Engine* 1823
- *Analytic Engine* 1833
  - The forerunner of modern digital computer!

Application

- Mathematical Tables – Astronomy
- Nautical Tables – Navy

Background

- Any continuous function can be approximated by a polynomial --- *Weierstrass*

Technology

- mechanical - gears, Jacquard's loom, simple calculators

Difference Engine

A machine to compute mathematical tables

Weierstrass:

- Any continuous function can be approximated by a polynomial
- Any polynomial can be computed from *difference* tables

An example

$f(n) = n^2 + n + 41$

$d1(n) = f(n) - f(n-1) = 2n$

$d2(n) = d1(n) - d1(n-1) = 2$

$f(n) = f(n-1) + d1(n) = f(n-1) + (d1(n-1) + 2)$

all you need is an adder!

n	0	1	2	3	4
d2(n)			2	2	2
d1(n)		2	4	6	8
f(n)	41	43	47	53	61

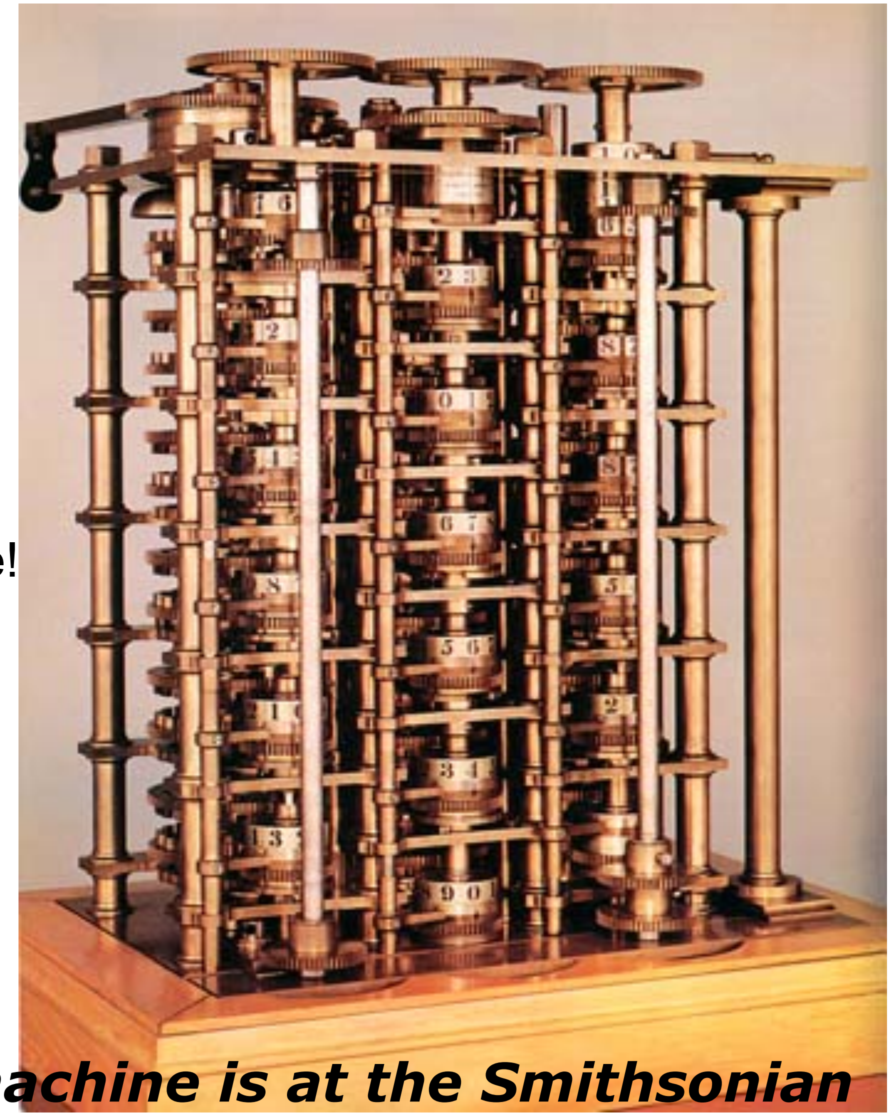
Difference Engine

1823
- Babbage's paper is published

1834
- The paper is read by Scheutz & his son in Sweden

1842
- Babbage gives up the idea of building it; he is onto Analytic Engine!

1855
- Scheutz displays his machine at the Paris World Fare
- Can compute any 6th degree polynomial
- *Speed:* 33 to 44 32-digit numbers per minute!



Now the machine is at the Smithsonian

Analytic Engine

1833: Babbage's paper was published

- *conceived during a hiatus in the development of the difference engine*

Inspiration: *Jacquard Looms*

- looms were controlled by punched cards
  - » The set of cards with fixed punched holes dictated the pattern of weave ⇒ *program*
  - » The same set of cards could be used with different colored threads ⇒ *numbers*

1871: Babbage dies

- The machine remains unrealized.

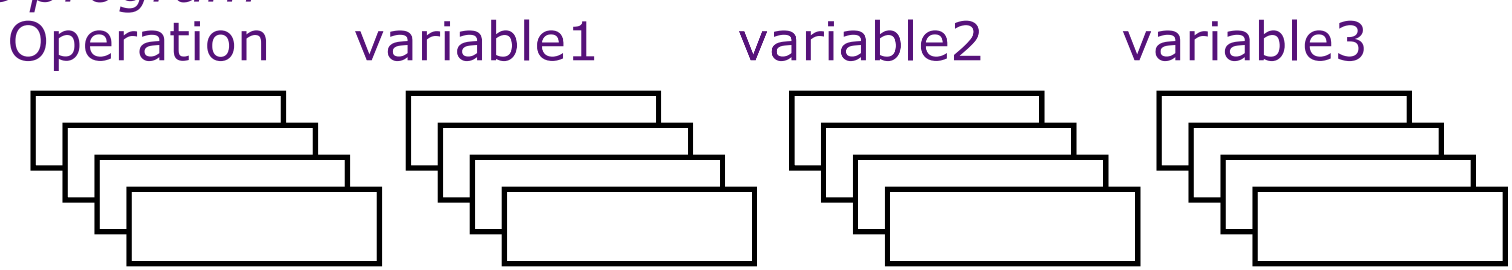
It is not clear if the analytic engine could be built even today using only mechanical technology

Analytic Engine

The first conception of a general-purpose computer

1. The *store* in which all variables to be operated upon, as well as all those quantities which have arisen from the results of the operations are placed.
2. The *mill* into which the quantities about to be operated upon are always brought.

The *program*



An operation in the *mill* required feeding two punched cards and producing a new punched card for the *store.*

An operation to alter the sequence was also provided!

The first programmer

Ada Byron *aka* "Lady Lovelace" 1815-52

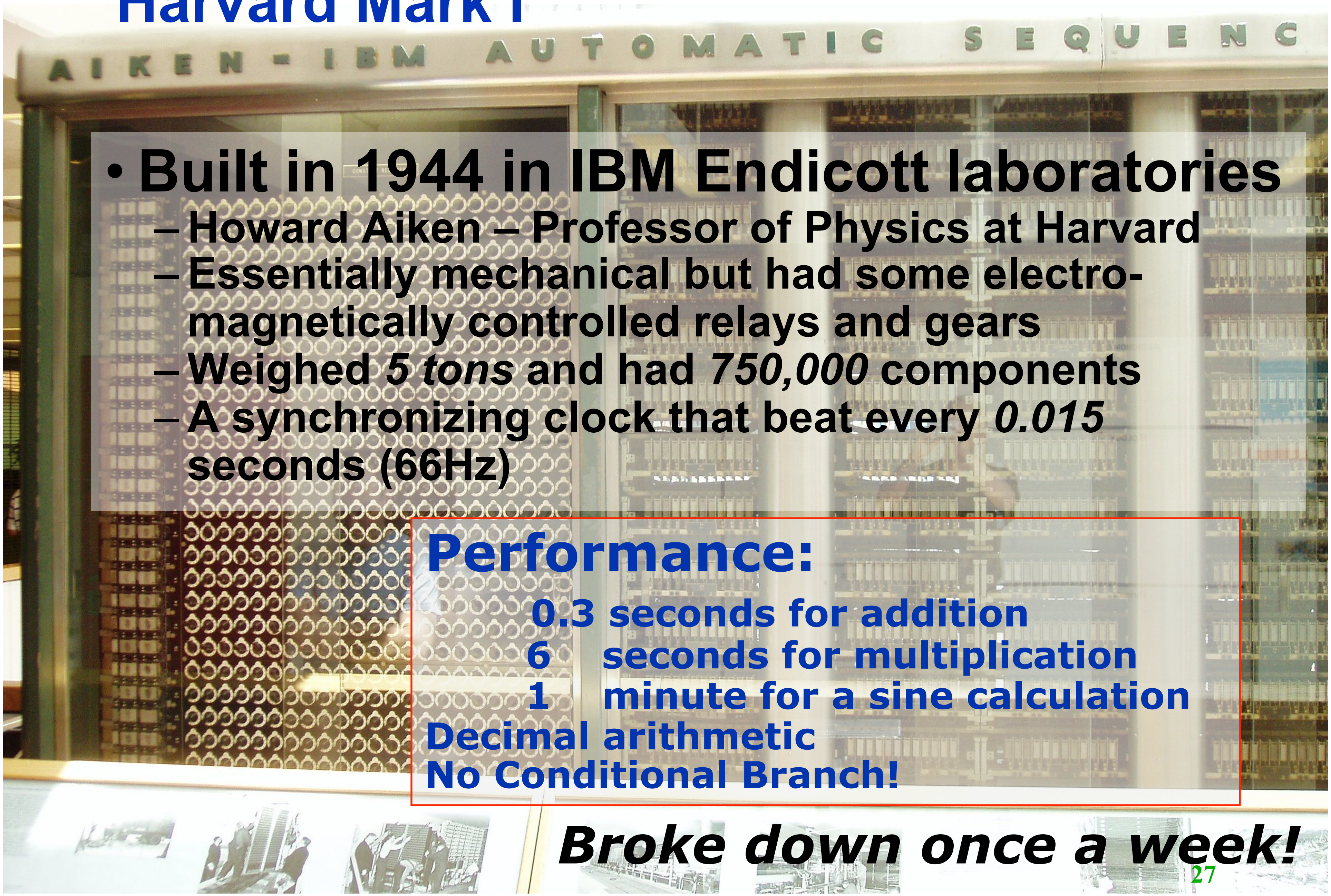
Ada's tutor was Babbage himself!

Babbage's Influence

- Babbage's ideas had great influence later primarily because of
  - *Luigi Menabrea,* who published notes of Babbage's lectures in Italy
  - *Lady Lovelace,* who translated Menabrea's notes in English and thoroughly expanded them.
    "... Analytic Engine weaves *algebraic patterns*...."

- In the early twentieth century - the focus shifted to analog computers but
  - *Harvard Mark I built in 1944 is very close in spirit to the Analytic Engine.*

Harvard Mark I



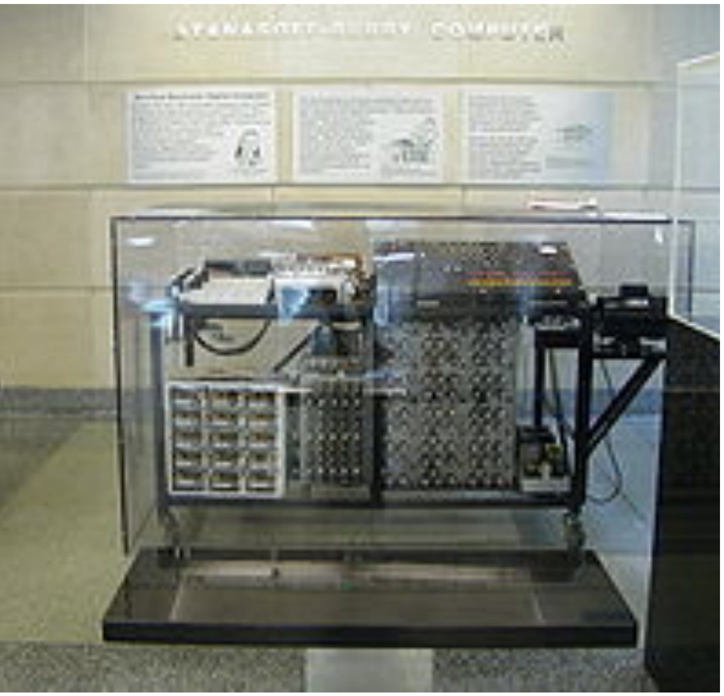
- **Built in 1944 in IBM Endicott laboratories**
  - **Howard Aiken – Professor of Physics at Harvard**
  - **Essentially mechanical but had some electro-magnetically controlled relays and gears**
  - **Weighed *5 tons* and had *750,000* components**
  - **A synchronizing clock that beat every *0.015* seconds (66Hz)**

Performance:

0.3 seconds for addition
6 seconds for multiplication
1 minute for a sine calculation
Decimal arithmetic
No Conditional Branch!

Broke down once a week!

Linear Equation Solver
John Atanasoff, Iowa State University



1930's:
- Atanasoff built the Linear Equation Solver.
- It had 300 tubes!
- Special-purpose binary digital calculator
- Dynamic RAM (stored values on refreshed capacitors)

Application:
- Linear and Integral differential equations

Background:
- Vannevar Bush's Differential Analyzer
  --- *an analog computer*

Technology:
- Tubes and Electromechanical relays

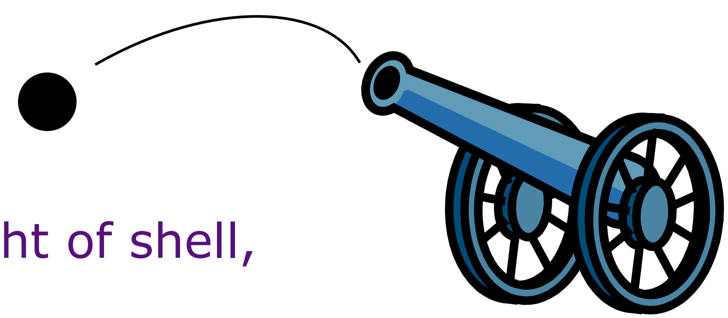
Atanasoff decided that the correct mode of computation was using electronic binary digits.

Electronic Numerical Integrator and Computer (ENIAC)

- Inspired by Atanasoff and Berry, Eckert and Mauchly designed and built ENIAC (1943-45) at the University of Pennsylvania
- The first, completely electronic, operational, general-purpose analytical calculator!
  - 30 tons, 72 square meters, 200KW
- Performance
  - Read in 120 cards per minute
  - Addition took 200 μs, Division 6 ms
  - 1000 times faster than Mark I
- Not very reliable!

WW-2 Effort

Application: Ballistic calculations



angle = f (location, tail wind, cross wind,
air density, temperature, weight of shell,
propellant charge, ...)

Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC's programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions "out of order"
- Eckert, Mauchly, John von Neumann and others designed EDVAC (1944) to solve this problem
  - Solution was the *stored program computer*

    ⇒ "*program can be manipulated as data*"
- *First Draft of a report on EDVAC* was published in 1945, but just had von Neumann's signature!
  - In 1973 the court of Minneapolis attributed the honor of *inventing the computer* to John Atanasoff

Stored Program Computer

Program = A sequence of instructions

How to control instruction sequencing?

manual control	calculators
automatic control	
external (paper tape)	Harvard Mark I , 1944 Zuse's Z1, WW2
internal	
plug board	ENIAC 1946
read-only memory	ENIAC 1948
read-write memory	EDVAC 1947 *(concept)*

- The same storage can be used to store program and data

EDSAC 1950 Maurice Wilkes

Technology Issues

ENIAC ⇒ EDVAC

ENIAC	EDVAC
18,000 tubes	4,000 tubes
20 10-digit numbers	2000 word storage mercury delay lines

ENIAC had many asynchronous parallel units but only one was active at a time

BINAC : Two processors that checked each other for reliability.

Didn't work well because processors never agreed

Dominant Problem: *Reliability*

Mean time between failures (MTBF)

MIT's Whirlwind with an MTBF of 20 min. was perhaps the most reliable machine !

Reasons for unreliability:

1. Vacuum Tubes

2. Storage medium
   acoustic delay lines
   mercury delay lines
   Williams tubes
   Selections

Reliability solved by invention of Core memory by J. Forrester 1954 at MIT for Whirlwind project

Commercial Activity: 1948-52

IBM's SSEC (follow on from Harvard Mark I)

Selective Sequence Electronic Calculator

- 150 word store.
- Instructions, constraints, and tables of data were read from paper tapes.
- 66 Tape reading stations!
- Tapes could be glued together to form a loop!
- Data could be output in one phase of computation and read in the next phase of computation.

And then there was IBM 701

IBM 701 -- 30 machines were sold in 1953-54

used CRTs as main memory, 72 tubes of 32x32b each

IBM 650 -- a cheaper, drum based machine,
more than 120 were sold in 1954
and there were orders for 750 more!

Users stopped building their own machines.

Why was IBM late getting into computer technology?

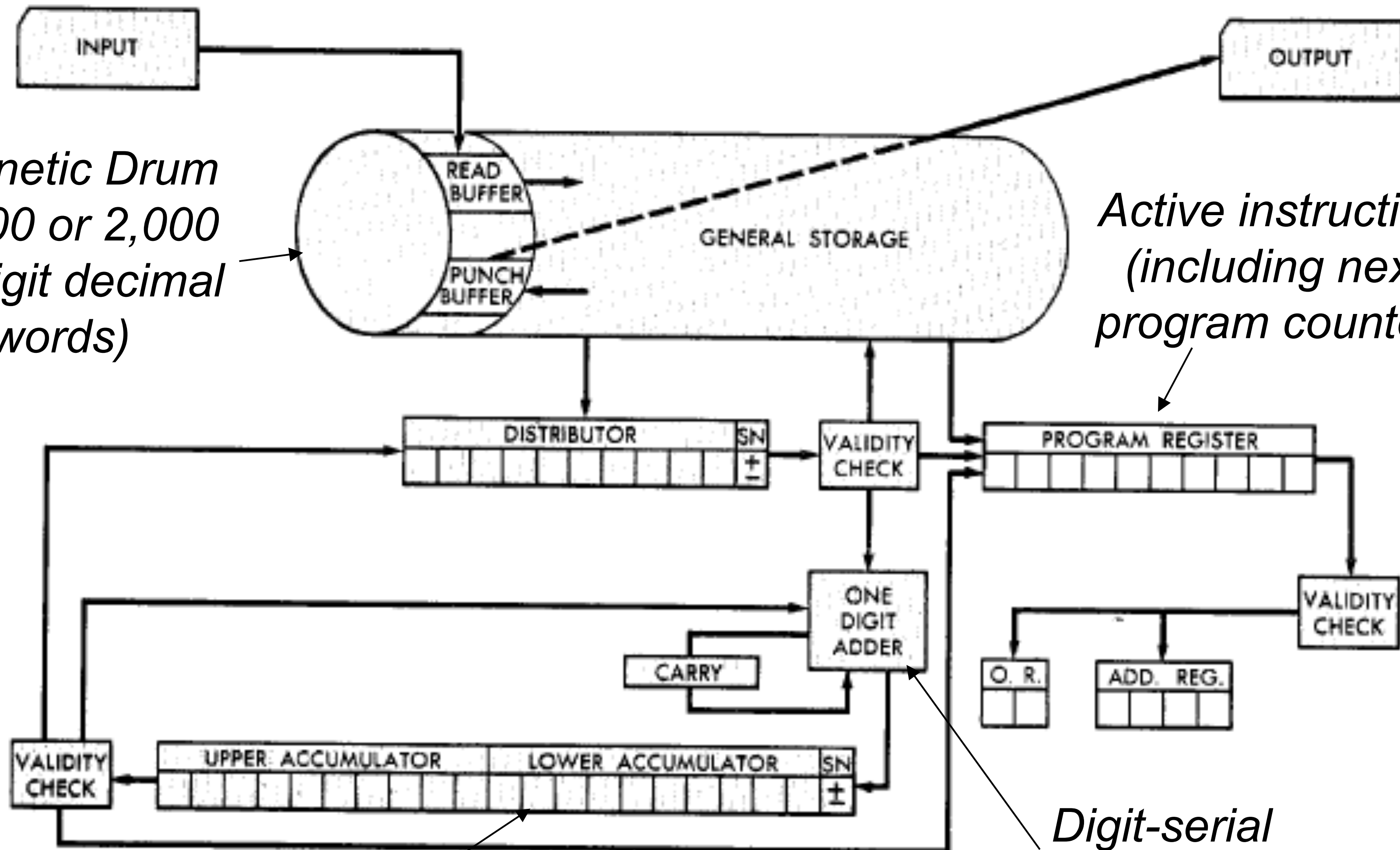
IBM was making too much money!

Even without computers, IBM revenues were doubling every 4 to 5 years in 40's and 50's.

Computers in mid 50's

- Hardware was expensive
- Stores were small (1000 words)
  - ⇒ No resident system software!
- Memory access time was 10 to 50 times slower than the processor cycle
  - ⇒ Instruction execution time was totally dominated by the *memory reference time*.
- The *ability to design complex control circuits* to execute an instruction was the central design concern as opposed to *the speed* of decoding or an ALU operation
- Programmer's view of the machine was inseparable from the actual hardware implementation

The IBM 650 (1953-4)



Magnetic Drum (1,000 or 2,000 10-digit decimal words)

Active instruction (including next program counter)

Digit-serial ALU

20-digit accumulator

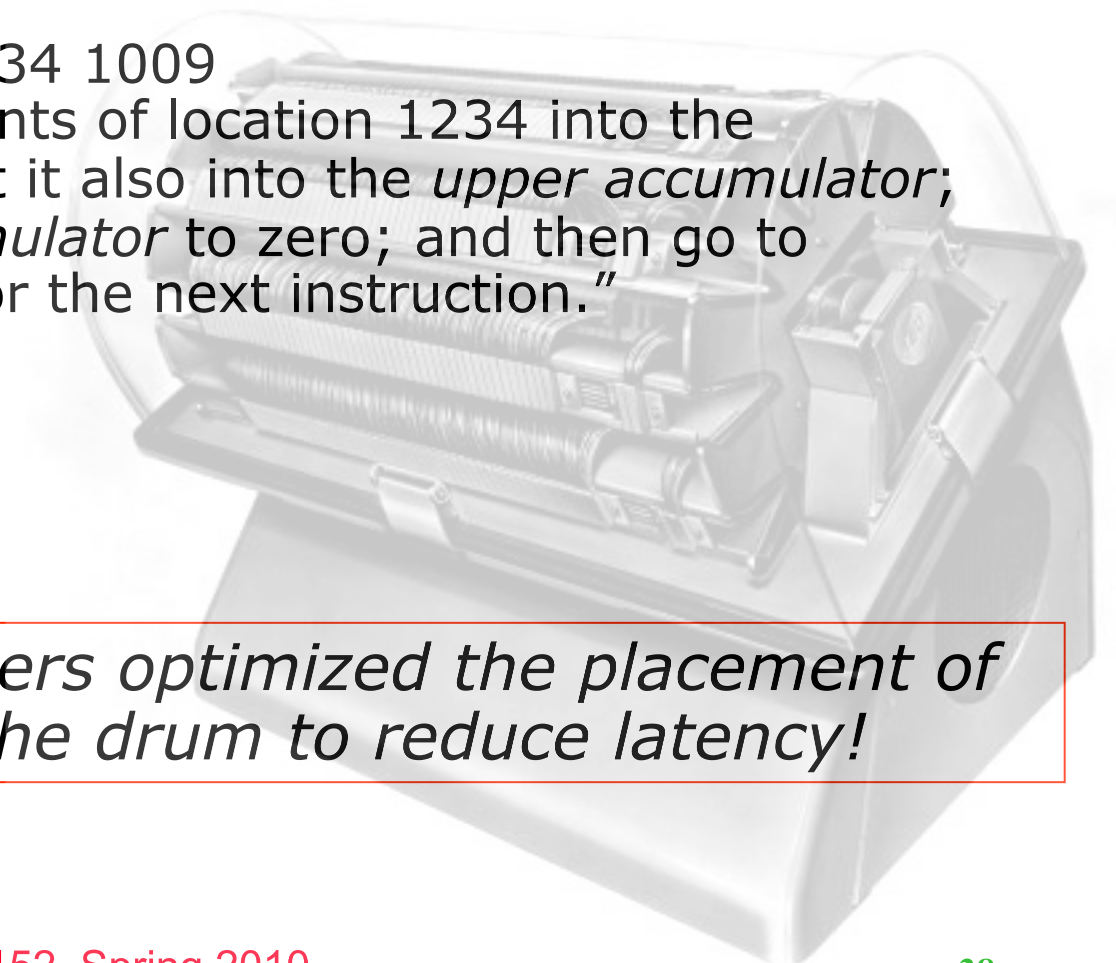
[From 650 Manual, © IBM]

Programmer's view of the IBM 650

A drum machine with 44 instructions

Instruction: 60 1234 1009

- "Load the contents of location 1234 into the *distribution*; put it also into the *upper accumulator*; set *lower accumulator* to zero; and then go to location 1009 for the next instruction."



Good programmers optimized the placement of instructions on the drum to reduce latency!

The Earliest Instruction Sets

Single Accumulator - A carry-over from the calculators.

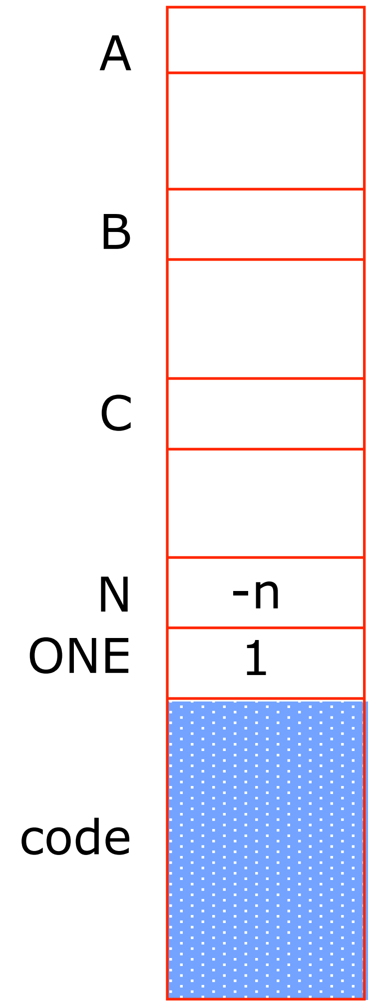
LOAD	x	AC ← M[x]
STORE	x	M[x] ← (AC)
ADD	x	AC ← (AC) + M[x]
SUB	x	
MUL	x	Involved a quotient register
DIV	x	
SHIFT LEFT		AC ← 2 × (AC)
SHIFT RIGHT		
JUMP	x	PC ← x
JGE	x	if (AC) ≥ 0 then PC ← x
LOAD ADR	x	AC ← Extract address field(M[x])
STORE ADR	x	

Typically less than 2 dozen instructions!

Programming: Single Accumulator Machine

$$C_i \leftarrow A_i + B_i, \quad 1 \le i \le n$$

```
LOOP    LOAD     N
        JGE      DONE
        ADD      ONE
        STORE    N
F1      LOAD     A
F2      ADD      B
F3      STORE    C
        JUMP     LOOP
DONE    HLT
```



How to modify the addresses A, B and C ?

Self-Modifying Code

$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$

```
LOOP  LOAD       N
      JGE        DONE
      ADD        ONE
      STORE      N
F1    LOAD       A
F2    ADD        B
F3    STORE      C
      LOAD ADR   F1
      ADD        ONE
      STORE ADR  F1
      LOAD ADR   F2
      ADD        ONE
      STORE ADR  F2
      LOAD ADR   F3
      ADD        ONE
      STORE ADR  F3
      JUMP       LOOP
DONE  HLT
```

modify the program for the next iteration (applies to the LOAD ADR F1 … STORE ADR F3 block)

Each iteration involves

	total	*book-keeping*
instruction fetches	*17*	*14*
operand fetches	*10*	*8*
stores	*5*	*4*

Index Registers

Tom Kilburn, Manchester University, mid 50's

One or more specialized registers to simplify address calculation

Modify existing instructions

LOAD	x, IX	AC ← M[x + (IX)]
ADD	x, IX	AC ← (AC) + M[x + (IX)]
...		

Add new instructions to manipulate *index registers*

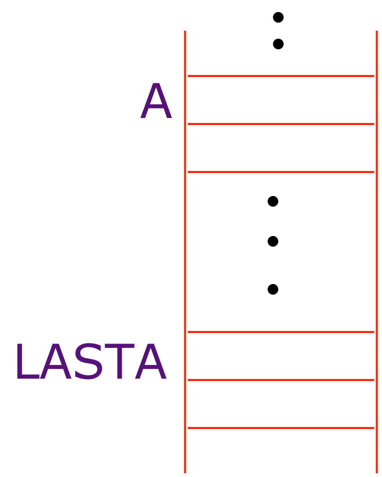
JZi	x, IX	if (IX)=0 *then* PC ← x
		else IX ← (IX) + 1
LOADi	x, IX	IX ← M[x] (truncated to fit IX)
...		

Index registers have accumulator-like characteristics

Using Index Registers

$C_i \leftarrow A_i + B_i, \quad 1 \le i \le n$

```
        LOADi   -n, IX
LOOP    JZi     DONE, IX
        LOAD    LASTA, IX
        ADD     LASTB, IX
        STORE   LASTC, IX
        JUMP    LOOP
DONE    HALT
```



- *Program does not modify itself*
- *Efficiency has improved dramatically (ops / iter)*

	with index regs	without index regs
instruction fetch	5(2)	17 (14)
operand fetch	2	10 (8)
store	1	5 (4)

- *Costs:* Instructions are 1 to 2 bits longer
  Index registers with ALU-like circuitry
  Complex control

Operations on Index Registers

To increment index register by k

$AC \leftarrow (IX)$ *new instruction*

$AC \leftarrow (AC) + k$

$IX \leftarrow (AC)$ *new instruction*

also the AC must be saved and restored.

It may be better to increment IX directly

INCi k, IX $IX \leftarrow (IX) + k$

More instructions to manipulate index register

STOREi x, IX $M[x] \leftarrow (IX)$ (extended to fit a word)

...

IX begins to look like an accumulator

$\Rightarrow$ several index registers

several accumulators

$\Rightarrow$ *General Purpose Registers*

Evolution of Addressing Modes

1. Single accumulator, absolute address

   LOAD x

2. Single accumulator, index registers

   LOAD x, IX

3. Indirection

   LOAD (x)

4. Multiple accumulators, index registers, indirection

   LOAD R, IX, x

   or LOAD R, IX, (x) the meaning?

   $R \leftarrow M[M[x] + (IX)]$

   or $R \leftarrow M[M[x + (IX)]]$

5. Indirect through registers

   LOAD R_I, (R_J)

6. The works

   LOAD R_I, R_J, (R_K) R_J = index, R_K = base addr

Variety of Instruction Formats

- *One address formats:* Accumulator machines
  - Accumulator is always other source and destination operand

- *Two address formats:* the destination is same as one of the operand sources

(Reg × Reg) to Reg	$R_I \leftarrow (R_I) + (R_J)$
(Reg × Mem) to Reg	$R_I \leftarrow (R_I) + M[x]$

  - *x* can be specified directly or via a register
  - effective address calculation for *x* could include indexing, indirection, ...

- *Three address formats:* One destination and up to two operand sources per instruction

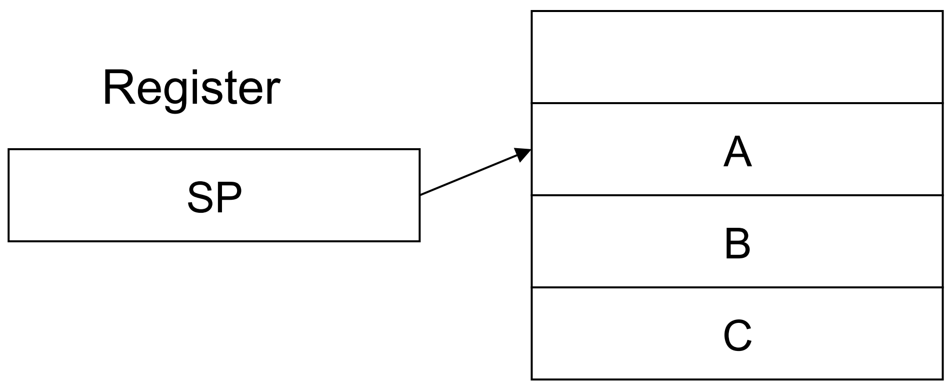
(Reg x Reg) to Reg	$R_I \leftarrow (R_J) + (R_K)$
(Reg x Mem) to Reg	$R_I \leftarrow (R_J) + M[x]$

Zero Address Formats

- Operands on a stack

  add   M[sp-1] ← M[sp] + M[sp-1]
  load   M[sp] ← M[M[sp]]

  – Stack can be in registers or in memory (usually top of stack cached in registers)



Burrough's B5000 Stack Architecture:
An ALGOL Machine, Robert Barton, 1960

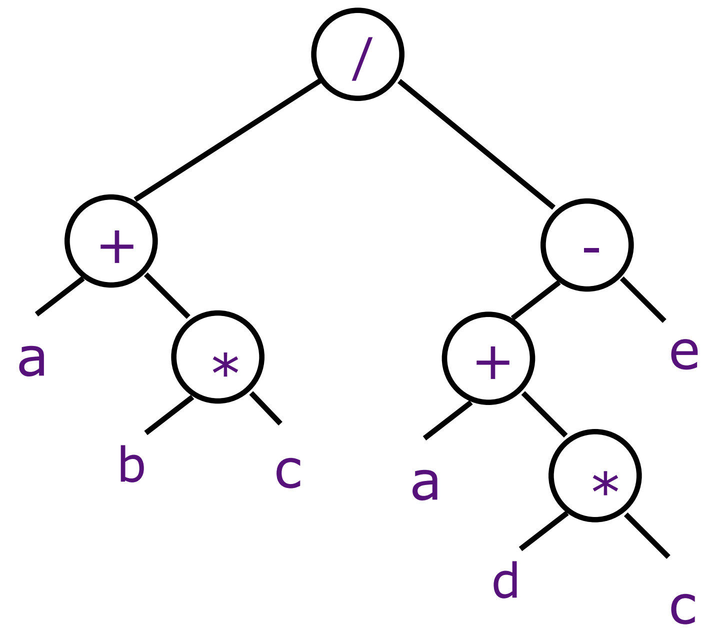
- Machine implementation can be completely hidden if the programmer is provided only a high-level language interface.

- *Stack machine* organization because stacks are convenient for:
  1. expression evaluation;
  2. subroutine calls, recursion, nested interrupts;
  3. accessing variables in block-structured languages.

- B6700, a later model, had many more innovative features
  - tagged data
  - virtual memory
  - multiple processors and memories

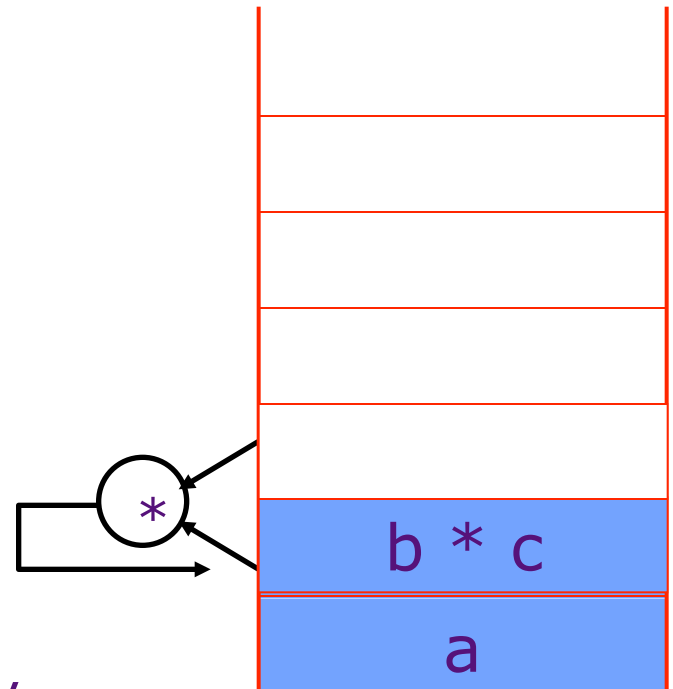
Evaluation of Expressions

(a + b * c) / (a + d * c - e)



Reverse Polish

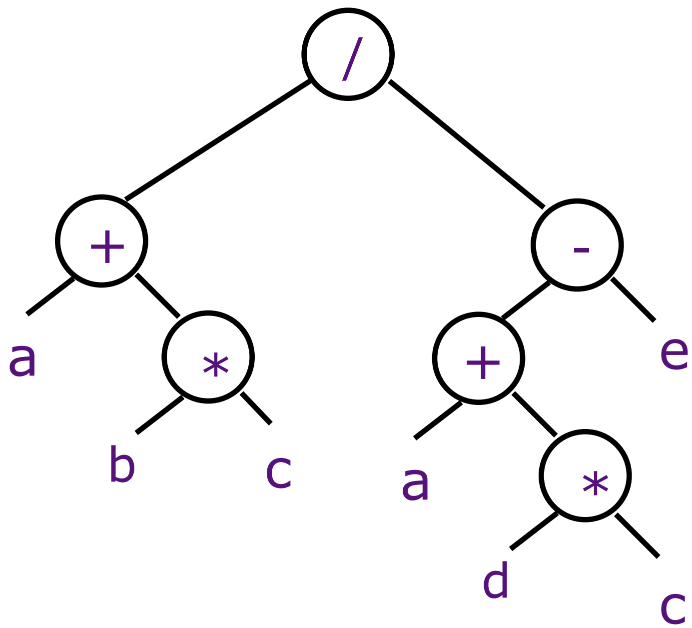
a b c * + a d c * + e - /



Evaluation Stack

Evaluation of Expressions

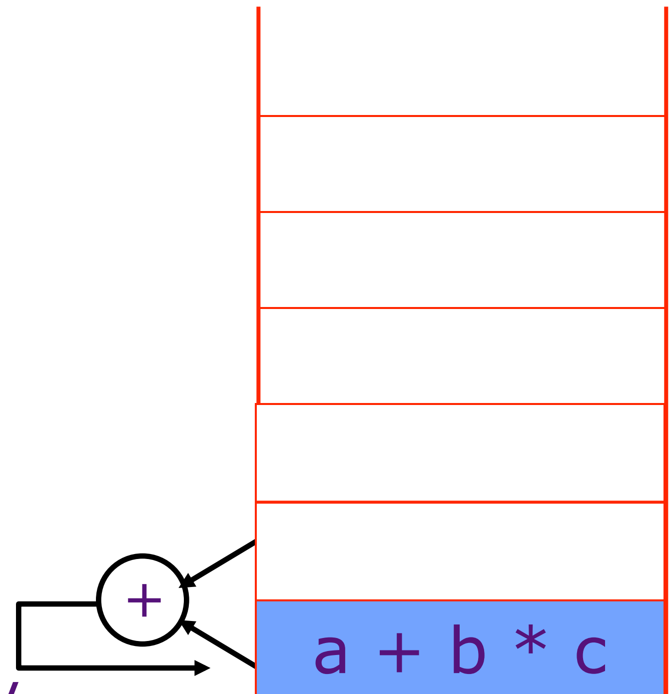
(a + b * c) / (a + d * c - e)



Reverse Polish

a b c * + a d c * + e - /

add



Evaluation Stack

Hardware organization of the stack

- Stack is part of the processor state
  ⇒ *stack must be bounded and small*
  ≈ number of Registers,
  not the size of main memory

- Conceptually stack is unbounded
  ⇒ *a part of the stack is included in the processor state; the rest is kept in the main memory*

Stack Operations and Implicit Memory References

- Suppose the top 2 elements of the stack are kept in registers and the rest is kept in the memory.

Each *push* operation $\Rightarrow$ 1 memory reference
pop operation $\Rightarrow$ 1 memory reference

No Good!

- Better performance can be got if the top N elements are kept in registers and memory references are made only when register stack overflows or underflows.

Issue - when to Load/Unload registers ?

Stack Size and Memory References

a b c * + a d c * + e - /

program	*stack (size = 2)*	*memory refs*
push a	R0	a
push b	R0 R1	b
push c	R0 R1 R2	c, ss(a)
*	R0 R1	sf(a)
+	R0	
push a	R0 R1	a
push d	R0 R1 R2	d, ss(a+b*c)
push c	R0 R1 R2 R3	c, ss(a)
*	R0 R1 R2	sf(a)
+	R0 R1	sf(a+b*c)
push e	R0 R1 R2	e,ss(a+b*c)
-	R0 R1	sf(a+b*c)
/	R0	

4 stores, 4 fetches (implicit)

Stack Size and Expression Evaluation

a b c * + a d c * + e - /

a and c are "loaded" twice

⇒

not the best use of registers!

program	*stack (size = 4)*
push a	R0
push b	R0 R1
push c	R0 R1 R2
*	R0 R1
+	R0
push a	R0 R1
push d	R0 R1 R2
push c	R0 R1 R2 R3
*	R0 R1 R2
+	R0 R1
push e	R0 R1 R2
-	R0 R1
/	R0

Register Usage in a GPR Machine

(a + b * c) / (a + d * c - e)

	Load	R0	a
	Load	R1	c
	Load	R2	b
Reuse R2	Mul	R2	R1
	Add	R2	R0
Reuse R3	Load	R3	d
	Mul	R3	R1
	Add	R3	R0
Reuse R0	Load	R0	e
	Sub	R3	R0
	Div	R2	R3

More control over register usage since registers can be named explicitly

Load	Ri m
Load	Ri (Rj)
Load	Ri (Rj) (Rk)

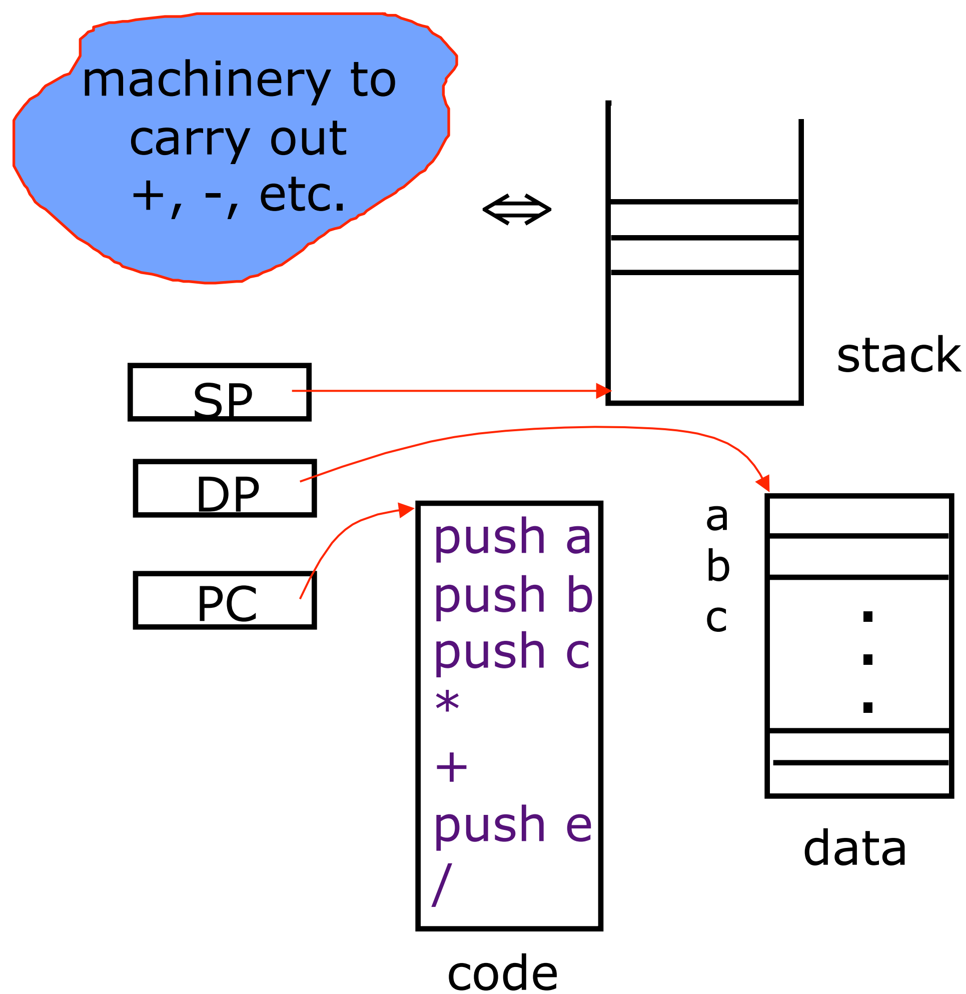
$\Rightarrow$

- *eliminates unnecessary Loads and Stores*
- *fewer Registers*

but instructions may be longer!

Stack Machines: Essential features

- In addition to push, pop, + etc., the instruction set must provide the capability to
  - *refer to any element in the data area*
  - *jump to any instruction in the code area*
  - *move any element in the stack frame to the top*



Stack versus GPR Organization

Amdahl, Blaauw and Brooks, 1964

1. The performance advantage of push down stack organization is derived from the presence of fast registers and not the way they are used.
2. "Surfacing" of data in stack which are "profitable" is approximately 50% because of constants and common subexpressions.
3. Advantage of instruction density because of implicit addresses is equaled if short addresses to specify registers are allowed.
4. Management of finite depth stack causes complexity.
5. Recursive subroutine advantage can be realized only with the help of an independent stack for addressing.
6. Fitting variable-length fields into fixed-width word is awkward.

Stack Machines (Mostly) Died by 1980

1. Stack programs are not smaller if short (Register) addresses are permitted.

2. Modern compilers can manage fast register space better than the stack discipline.

GPR's and caches are better than stack and displays

Early language-directed architectures often did not take into account the role of compilers!

B5000, B6700, HP 3000, ICL 2900, Symbolics 3600

Some would claim that an echo of this mistake is visible in the SPARC architecture register windows - more later…

Stacks post-1980

- Inmos Transputers (1985-2000)
  - Designed to support many parallel processes in Occam language
  - Fixed-height stack design simplified implementation
  - Stack trashed on context swap (fast context switches)
  - Inmos T800 was world's fastest microprocessor in late 80's
- Forth machines
  - Direct support for Forth execution in small embedded real-time environments
  - Several manufacturers (Rockwell, Patriot Scientific)
- Java Virtual Machine
  - Designed for software emulation, not direct hardware execution
  - Sun PicoJava implementation + others
- Intel x87 floating-point unit
  - Severely broken stack model for FP arithmetic
  - Deprecated in Pentium-4, replaced with SSE2 FP registers

Software Developments

up to 1955 Libraries of numerical routines

- Floating point operations
- Transcendental functions
- Matrix manipulation, equation solvers, . . .

1955-60 *High level Languages* - Fortran 1956
Operating Systems -

- Assemblers, Loaders, Linkers, Compilers
- Accounting programs to keep track of usage and charges

Machines required *experienced operators*

⇒ Most users could not be expected to understand these programs, much less write them

⇒ Machines had to be sold with a lot of resident software

Compatibility Problem at IBM

By early 60's, *IBM had 4 incompatible lines of computers!*

701	→	7094
650	→	7074
702	→	7080
1401	→	7010

Each system had its own

- Instruction set
- I/O system and Secondary Storage:
  magnetic tapes, drums and disks
- assemblers, compilers, libraries,...
- market niche
  business, scientific, real time, ...

⇒ *IBM 360*

IBM 360 : Design Premises

Amdahl, Blaauw and Brooks, 1964

- The design must lend itself to *growth and successor machines*
- General method for connecting I/O devices
- Total performance - answers per month rather than bits per microsecond $\Rightarrow$ *programming aids*
- Machine must be capable of *supervising itself* without manual intervention
- Built-in *hardware fault checking* and locating aids to reduce down time
- Simple to assemble systems with redundant I/O devices, memories etc. for *fault tolerance*
- Some problems required floating-point larger than 36 bits

IBM 360: *A General-Purpose Register (GPR) Machine*

- Processor State
  - 16 General-Purpose 32-bit Registers
    - » *may be used as index and base register*
    - » *Register 0 has some special properties*
  - 4 Floating Point 64-bit Registers
  - A Program Status Word (PSW)
    - » *PC, Condition codes, Control flags*
- A 32-bit machine with 24-bit addresses
  - But no instruction contains a 24-bit address!
- Data Formats
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words

The IBM 360 is why bytes are 8-bits long today!

IBM 360: Initial Implementations

	Model 30	. . .	Model 70
Storage	8K - 64 KB		256K - 512 KB
Datapath	8-bit		64-bit
Circuit Delay	30 nsec/level		5 nsec/level
Local Store	Main Store		Transistor Registers
Control Store	Read only 1μsec		Conventional circuits

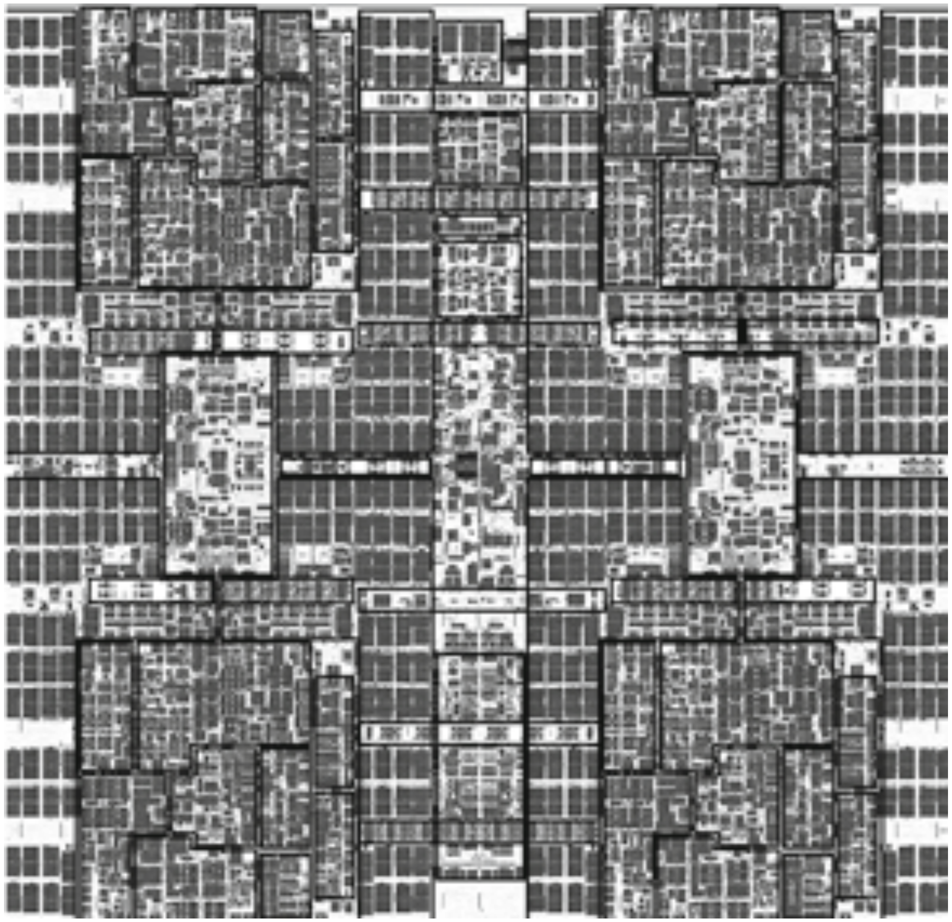
IBM 360 instruction set architecture (ISA) completely hid the underlying technological differences between various models.

Milestone: The first true ISA designed as portable hardware-software interface!

With minor modifications it still survives today!

IBM 360: 45 years later… The zSeries z10 Microprocessor

- 4.4 GHz in IBM 65nm SOI CMOS technology
- 994 million transistors in 454mm^2
- 64-bit virtual addressing
  - original S/360 was 24-bit, and S/370 was 31-bit extension
- Quad core design
- Dual-issue in-order superscalar CISC pipeline
- Out-of-order memory accesses
- Redundant datapaths
  - every instruction performed in two parallel datapaths and results compared
- 64KB L1 I-cache, 128KB L1 D-cache on-chip
- 3MB private L2 unified cache per core, on-chip
- Off-chip L3 cache of up to 48MB
- 10K-entry Branch Target Buffer
  - Very large buffer to support commercial workloads
- Hardware for decimal floating-point arithmetic
  - Important for business applications



[IEEE Micro, March 2008]

And in conclusion …

- Computer Architecture >> ISAs and RTL
- CS152 is about interaction of hardware and software, and design of appropriate abstraction layers
- Computer architecture is shaped by technology and applications
  - History provides lessons for the future
- Computer Science at the crossroads from sequential to parallel computing
  - Salvation requires innovation in many fields, including computer architecture
- Read Chapter 1, then Appendix B for next time!

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252