CS 61C
Great Ideas in Computer Architecture (a.k.a. Machine Structures)
Lecture 1: *Course Introduction*

Instructors:

Bernhard Boser

Randy H. Katz

http://inst.eecs.berkeley.edu/~cs61c/

Agenda

- Thinking about Machine Structures
- Great Ideas in Computer Architecture
- What You Need to Know About This Class
- Everything is a Number

Agenda

- Thinking about Machine Structures
- Great Ideas in Computer Architecture
- What You Need to Know About This Class
- Everything is a Number

Most Popular Programming Languages

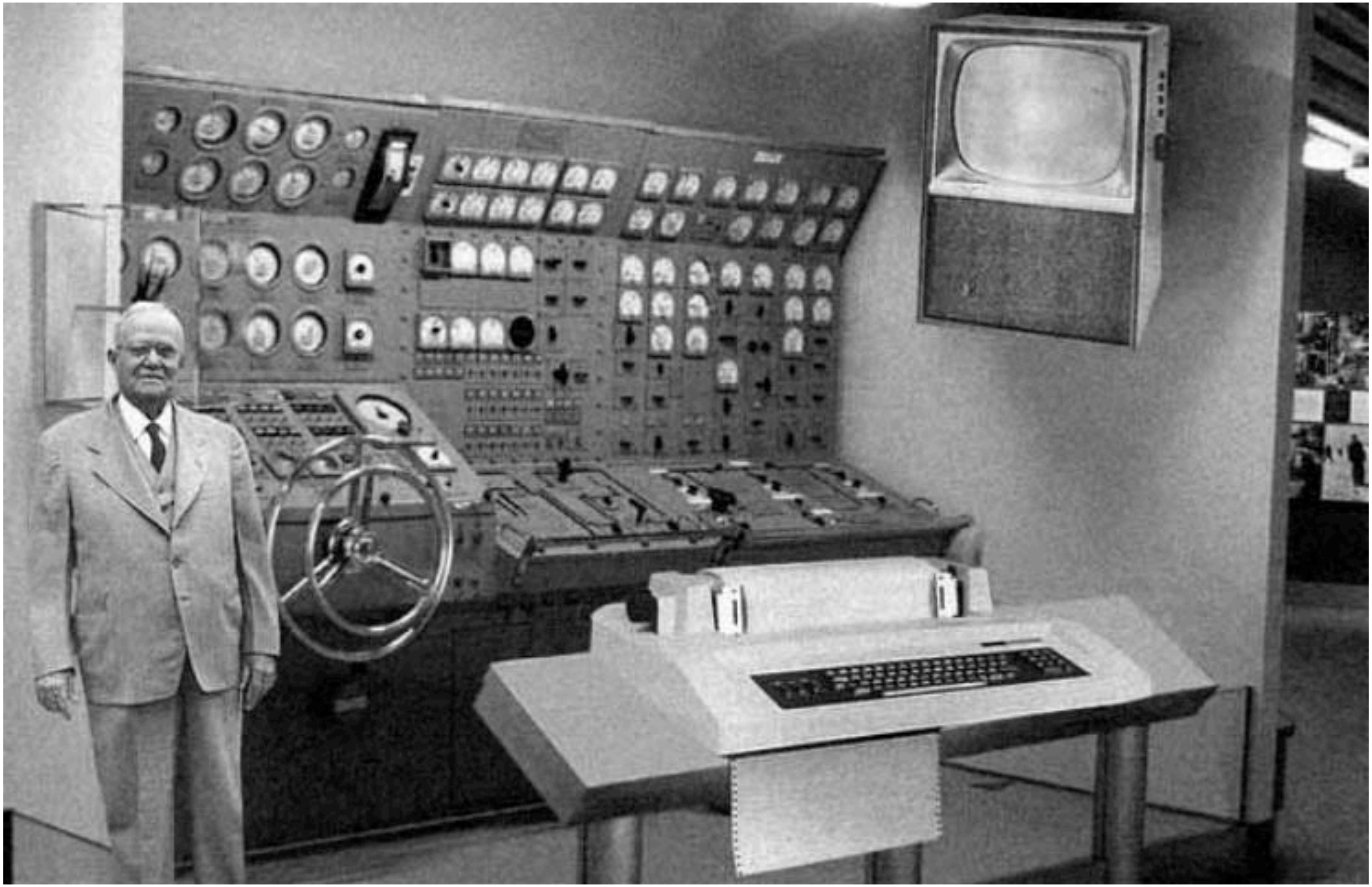
Language Rank	Types	Spectrum Ranking
1. C	mobile, desktop, embedded	100.0
C is used to write software where speed and flexibility is important, such as in embedded systems or high-performance computing.		
2. Java	web, mobile, desktop	98.1
Designed to allow the creation of programs that can run on different platforms with little or no modification, Java is a popular choice for Web applications.		
3. Python	web, desktop	98.0
A scripting language that is often used by software developers to add programmability to their applications, such as engineering-analysis tools or animation software.		
4. C++	mobile, desktop, embedded	95.9
5. R	desktop	87.9
6. C#	web, mobile, desktop	86.7
7. PHP	web	82.8
8. JavaScript	web, mobile	82.2
9. Ruby	web, desktop	74.5
10. Go	web, desktop	71.9
11. Swift	mobile, desktop	70.1
12. Arduino	embedded	69.9

Why You Need to Learn C!

CS61C is NOT really about C Programming

- It is about the *hardware-software interface*
  - What does the programmer need to know to achieve the highest possible performance
- C is close to the underlying hardware, unlike languages like Python and Java!
  - Allows us to talk about key hardware features in higher level terms
  - Allows programmer to explicitly harness underlying hardware parallelism for high performance

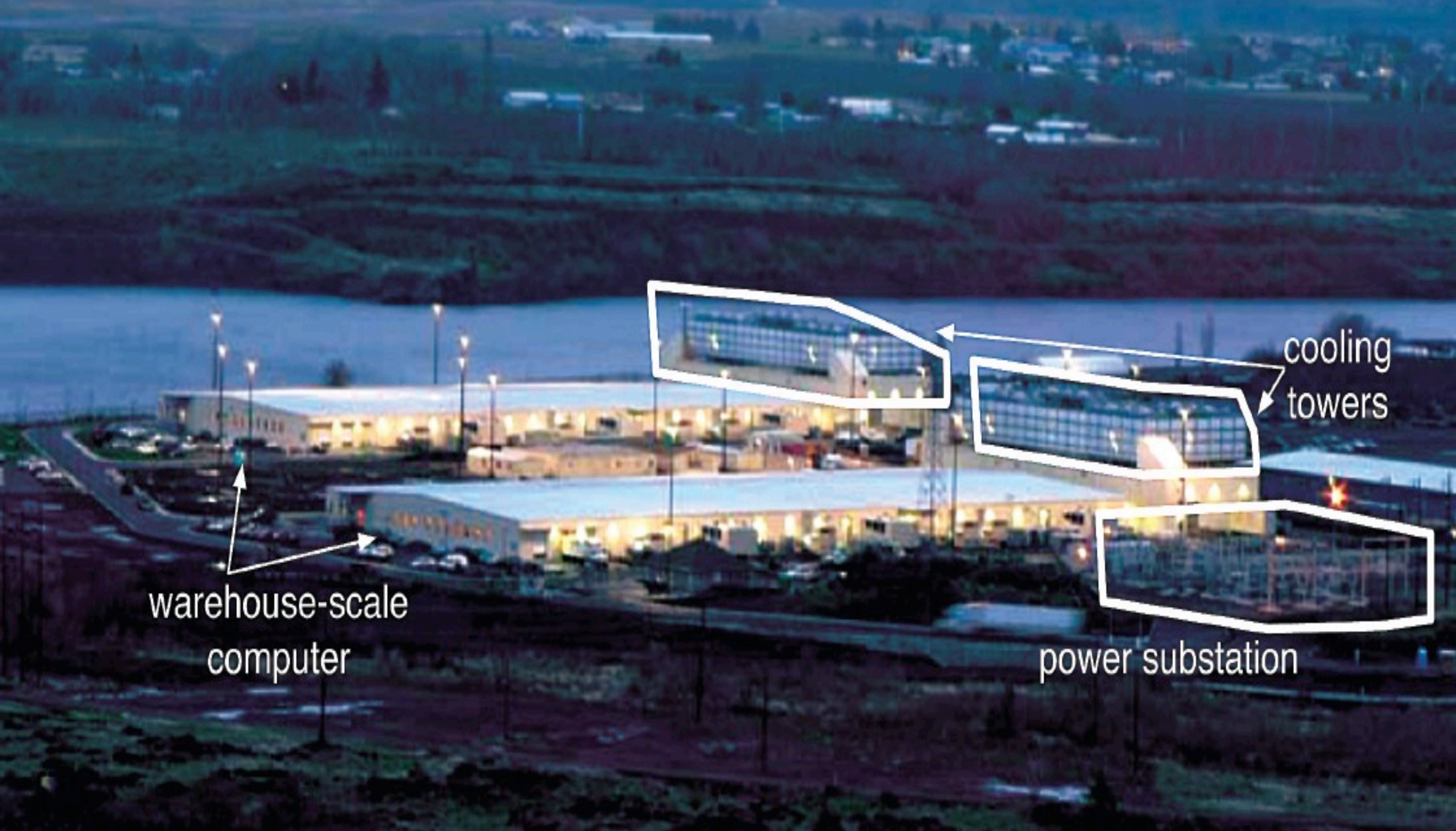
Old School CS61C



New School CS61C (1/2)

Personal
Mobile
Devices

New School CS61C (2/2)



9

New-School Machine Structures

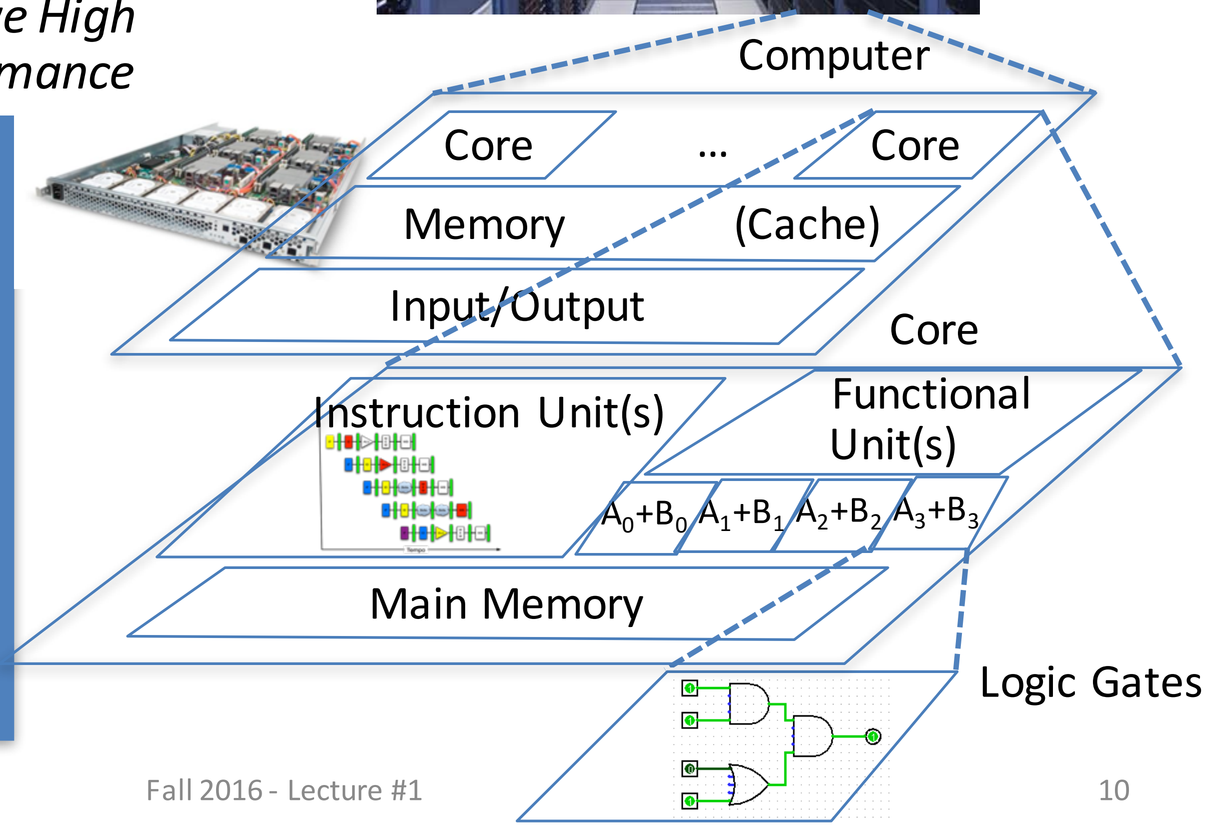
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "cats"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates functioning in parallel at same time

Harness Parallelism & Achieve High Performance

Warehouse-Scale Computer

Smart Phone

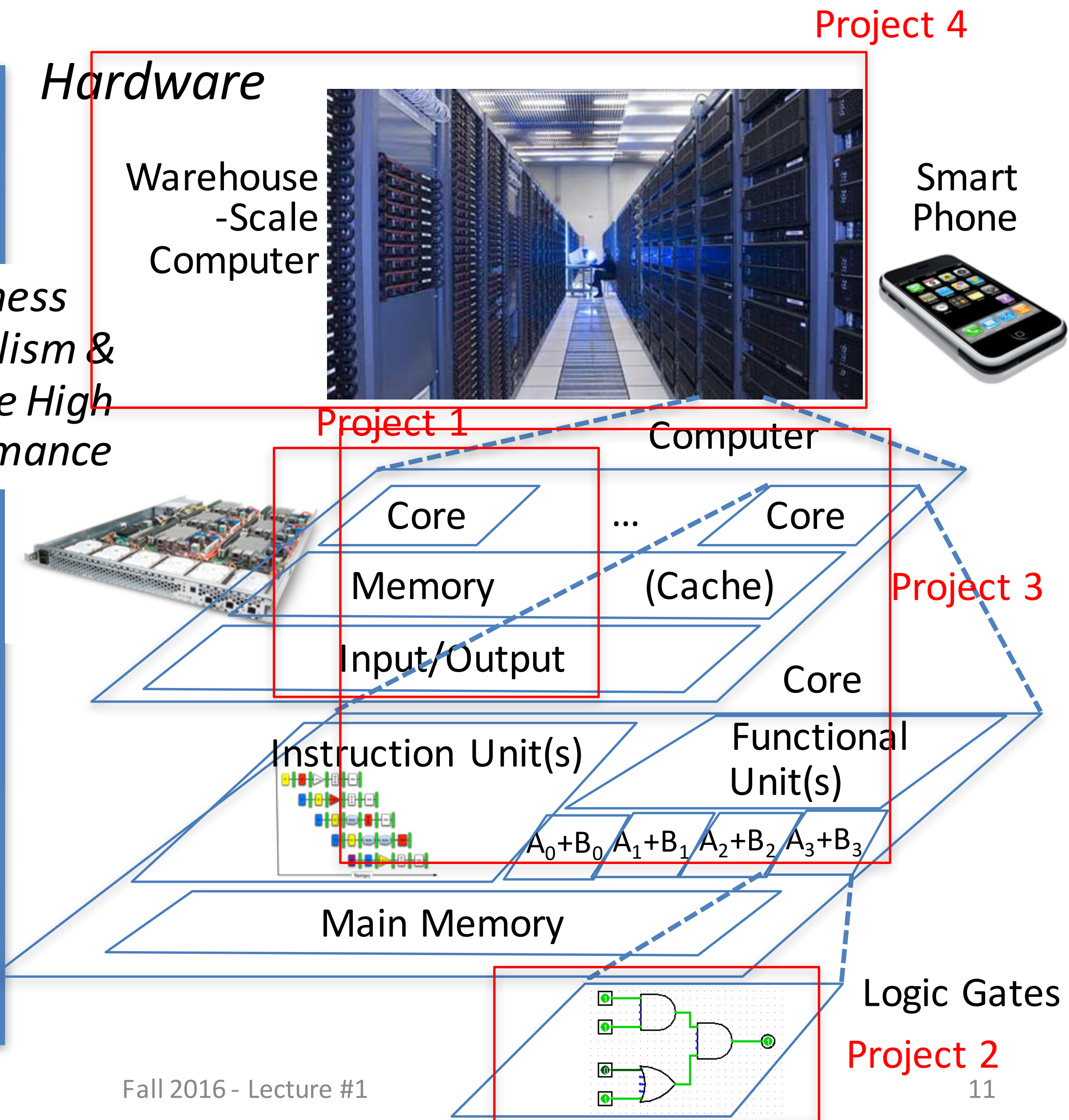


New-School Machine Structures

Software | *Hardware*

- **Parallel Requests**
  Assigned to computer
  e.g., Search "cats"
- **Parallel Threads**
  Assigned to core
  e.g., Lookup, Ads
- **Parallel Instructions**
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- **Parallel Data**
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- **Hardware descriptions**
  All gates functioning in parallel at same time

Harness Parallelism & Achieve High Performance



Agenda

- Thinking about Machine Structures
- Great Ideas in Computer Architecture
- What You Need to Know About This Class
- Everything is a Number

Five Great Ideas in Computer Architecture

1. Abstraction
   (Layers of Representation/Interpretation)
2. Moore's Law (Designing through trends)
3. Principle of Locality (Memory Hierarchy)
4. Parallelism
5. Dependability via Redundancy

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

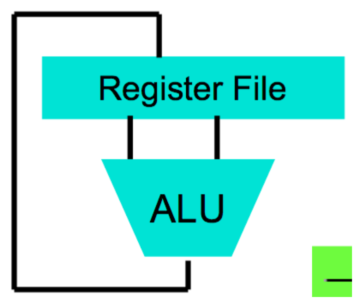
Logic Circuit Description (Circuit Schematic Diagrams)

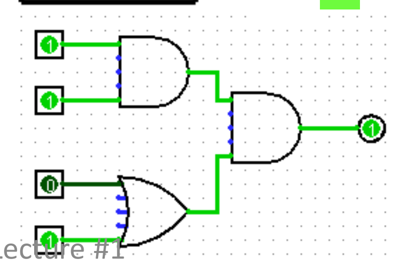
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101                1
0101 1000 0000 1001 1100                1
```





#2: Moore's Law

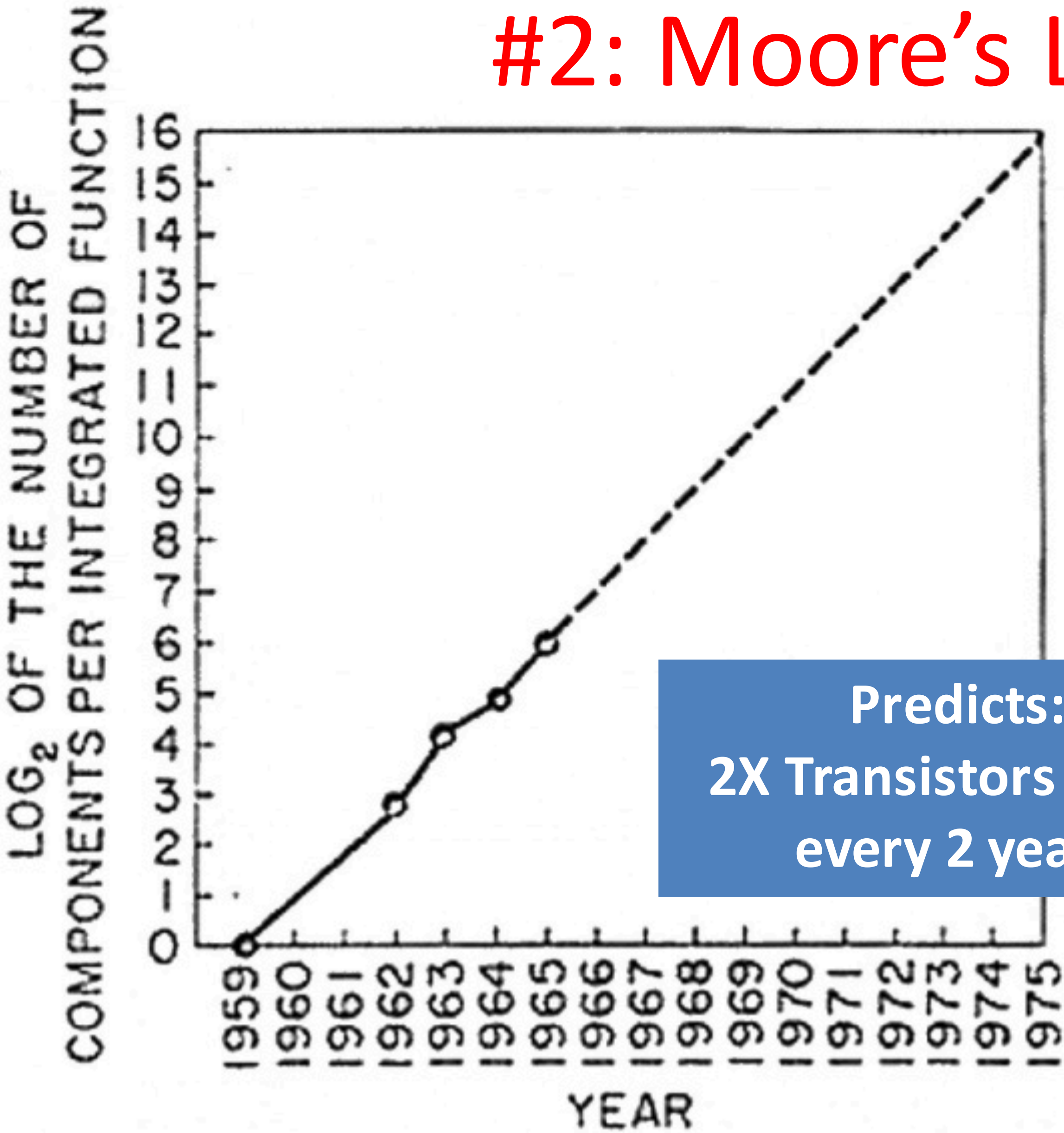
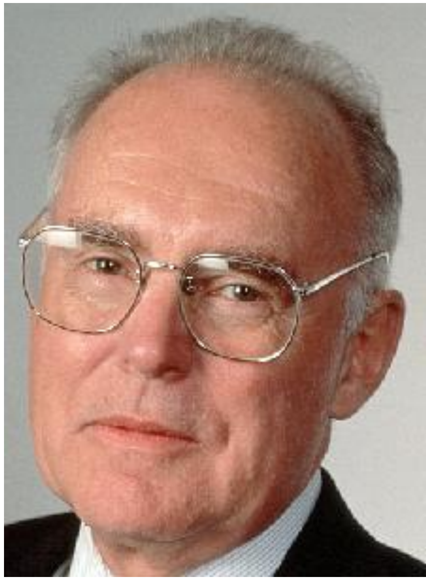
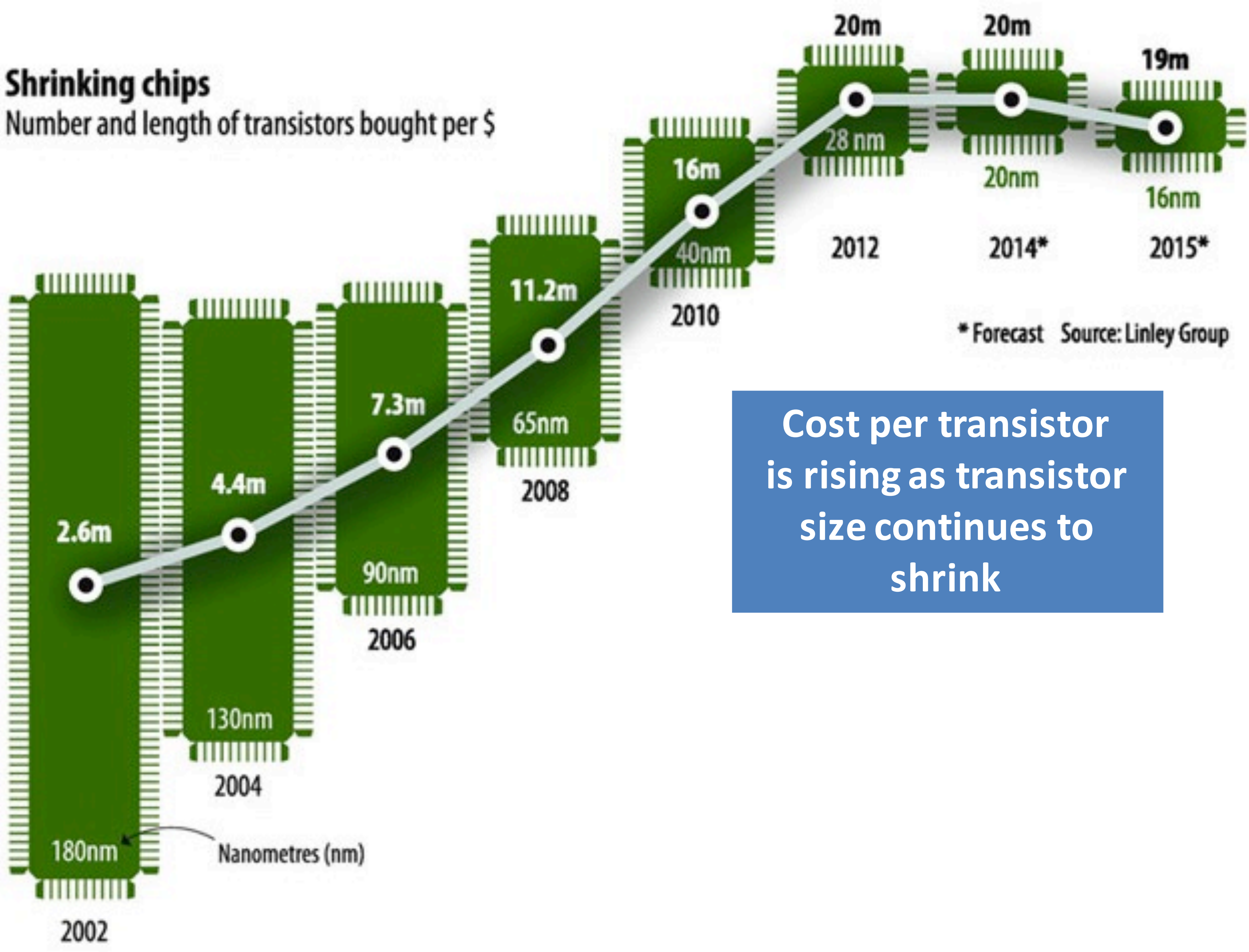


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Predicts:
2X Transistors / chip every 2 years



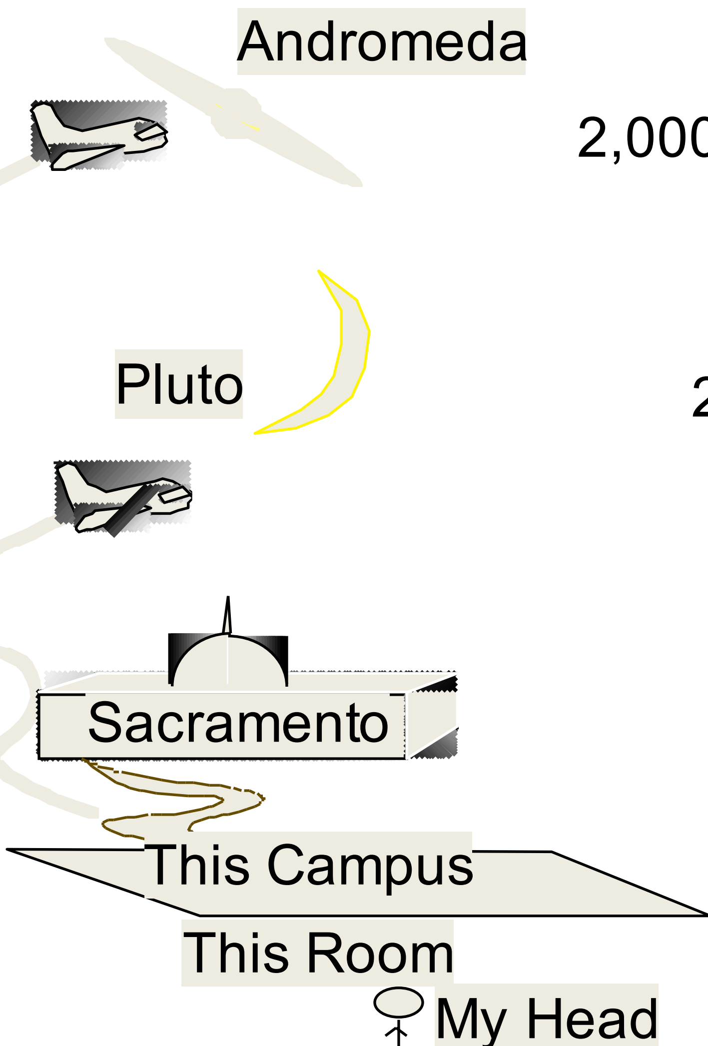
Gordon Moore
Intel Cofounder
B.S. Cal 1950!

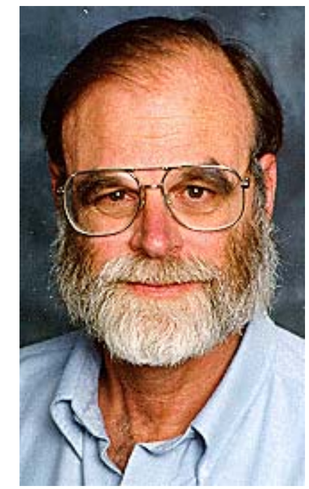


Cost per transistor is rising as transistor size continues to shrink

Jim Gray's Storage Latency Analogy: How Far Away is the Data?

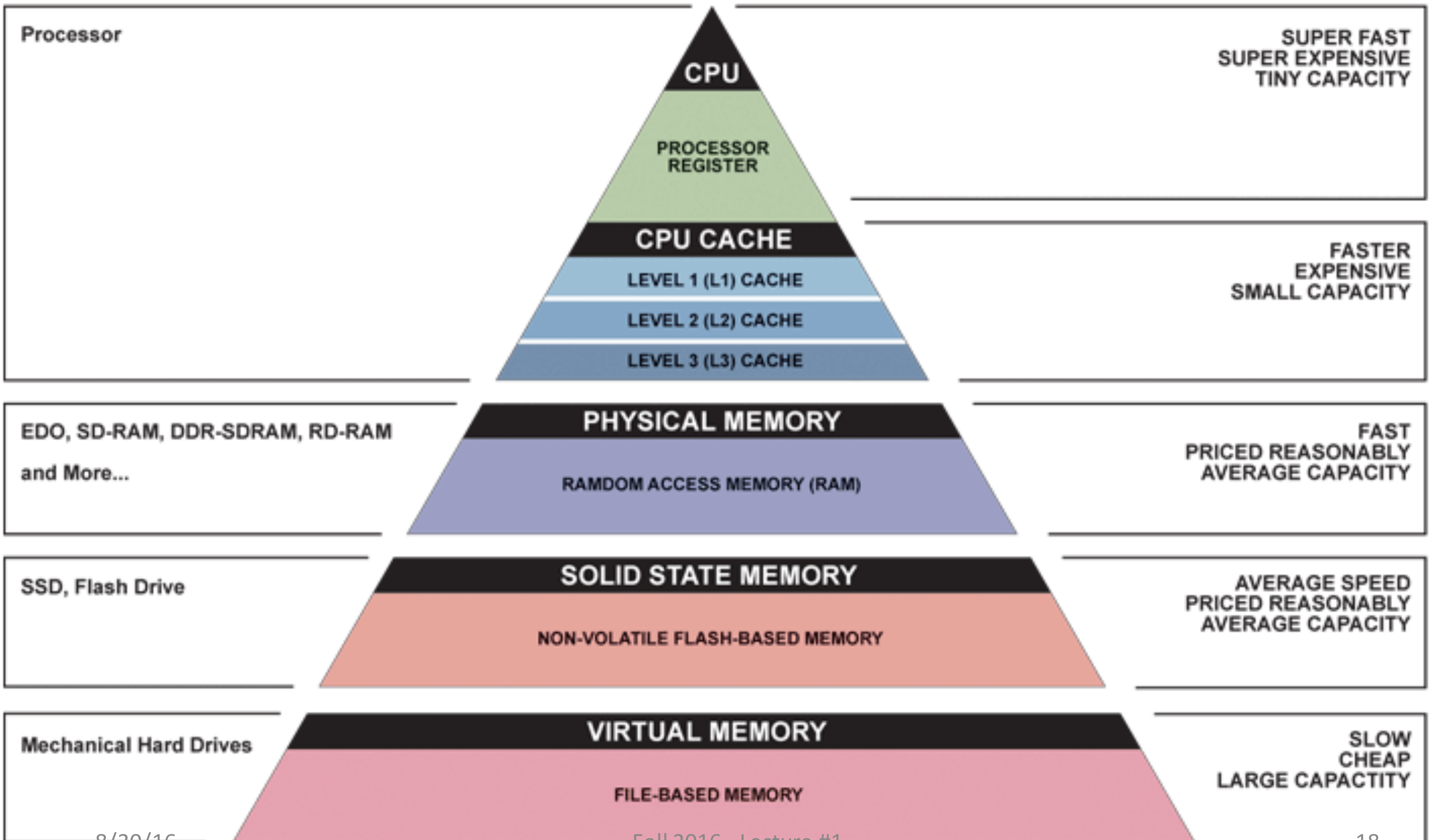
(ns)		
10^9	Tape /Optical Robot	2,000 Years
10^6	Disk	2 Years
100	Main Memory	1.5 hr
10	On Board Cache	10 min
2	On Chip Cache	
1	Registers	1 min



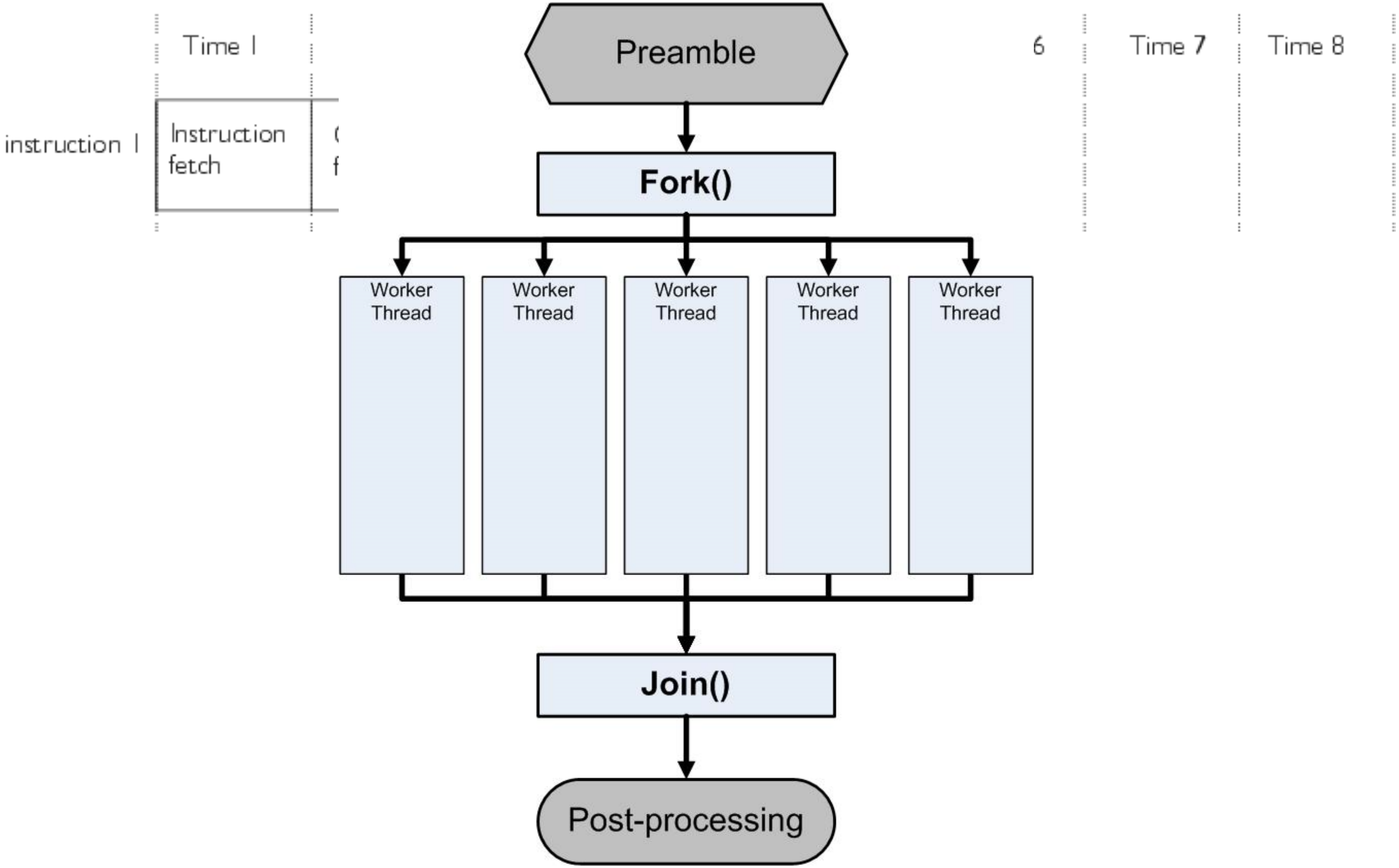


Jim Gray
Turing Award
B.S. Cal 1966
Ph.D. Cal 1969!

Great Idea #3: Principle of Locality/ Memory Hierarchy



Great Idea #4: Parallelism



Caveat: Amdahl's Law

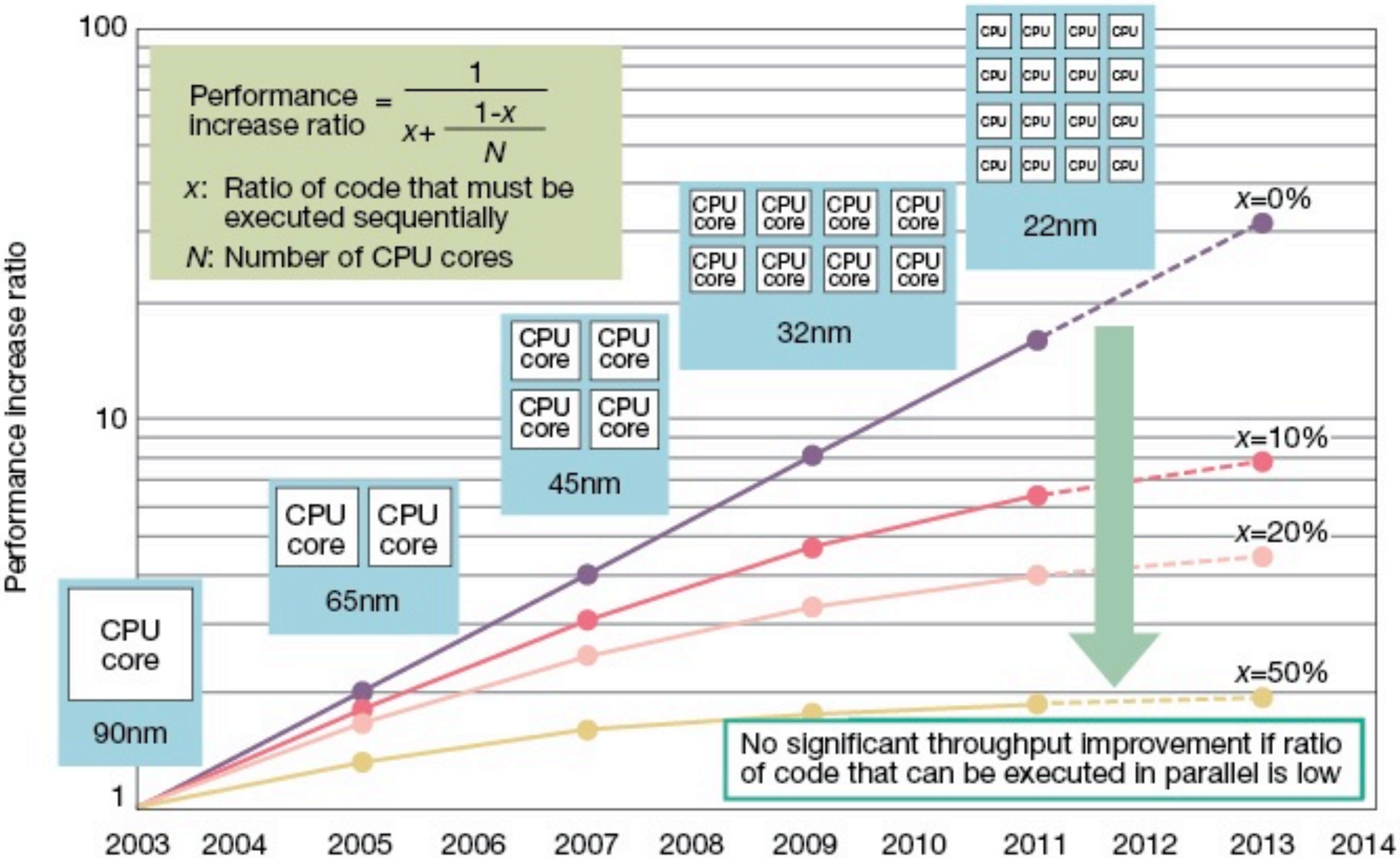


Fig 3 Amdahl's Law an Obstacle to Improved Performance Performance will not rise in the same proportion as the increase in CPU cores. Performance gains are limited by the ratio of software processing that must be executed sequentially. Amdahl's Law is a major obstacle in boosting multicore microprocessor performance. Diagram assumes no overhead in parallel processing. Years shown for design rules based on Intel planned and actual technology. Core count assumed to double for each rule generation.

Gene Amdahl
Computer Pioneer

Coping with Failures

- 4 disks/server, 50,000 servers
- Failure rate of disks: 2% to 10% / year
  - Assume 4% annual failure rate
- On average, how often does a disk fail?
  a) 1 / month
  b) 1 / week
  c) 1 / day
  d) 1 / hour

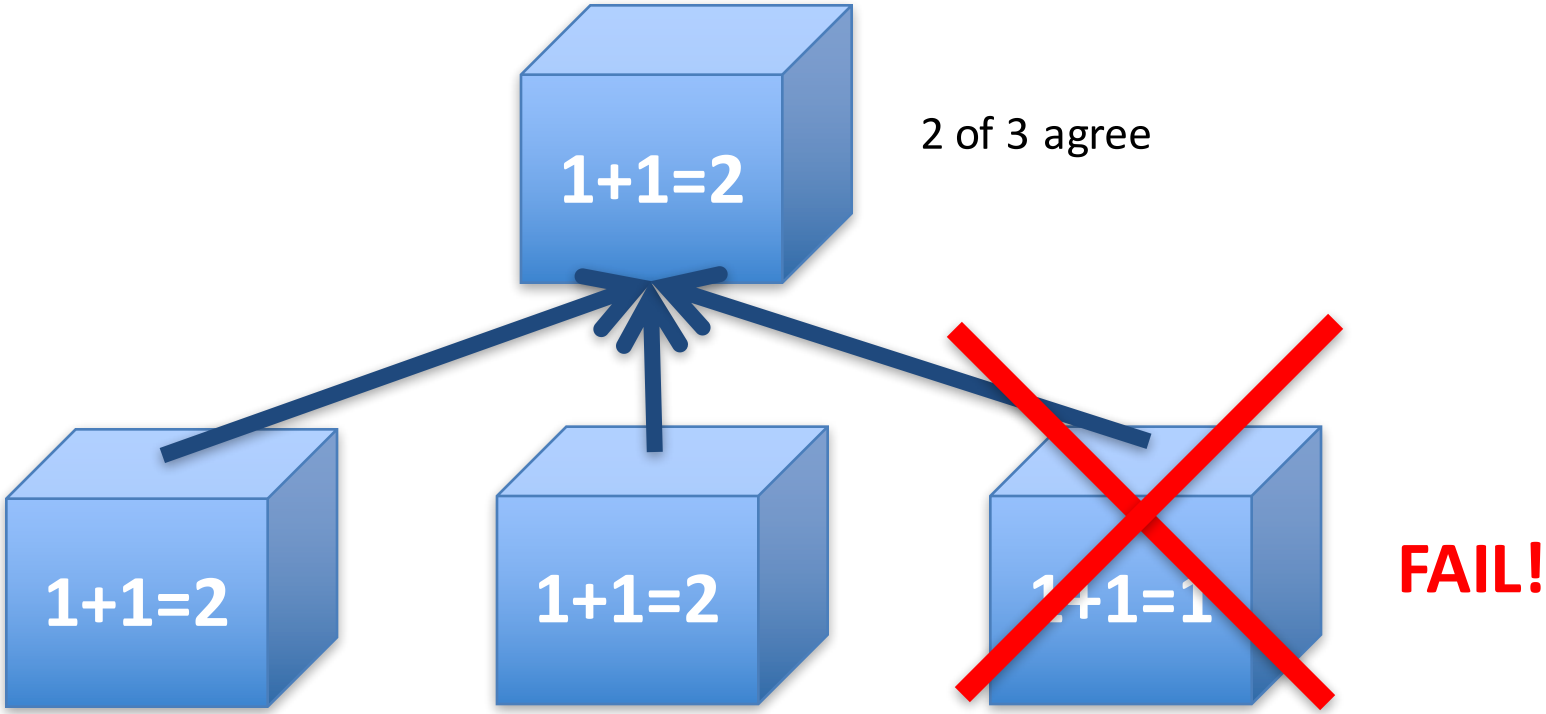
Coping with Failures

- 4 disks/server, 50,000 servers
- Failure rate of disks: 2% to 10% / year
  - Assume 4% annual failure rate
- On average, how often does a disk fail?
  a) 1 / month
  b) 1 / week
  c) 1 / day
  d) 1 / hour

50,000 x 4 = 200,000 disks
200,000 x 4% = 8000 disks fail
365 days x 24 hours = 8760 hours

Great Idea #5: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail



Increasing transistor density reduces the cost of redundancy

Great Idea #5: Dependability via Redundancy

- Applies to everything from datacenters to storage to memory to instructors
  - Redundant datacenters so that can lose 1 datacenter but Internet service stays online
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Your Turn

- You have 8 disks, 1TB each.
  - What is the minimum number of extra disks to insure that no information is lost if a single disk fails?

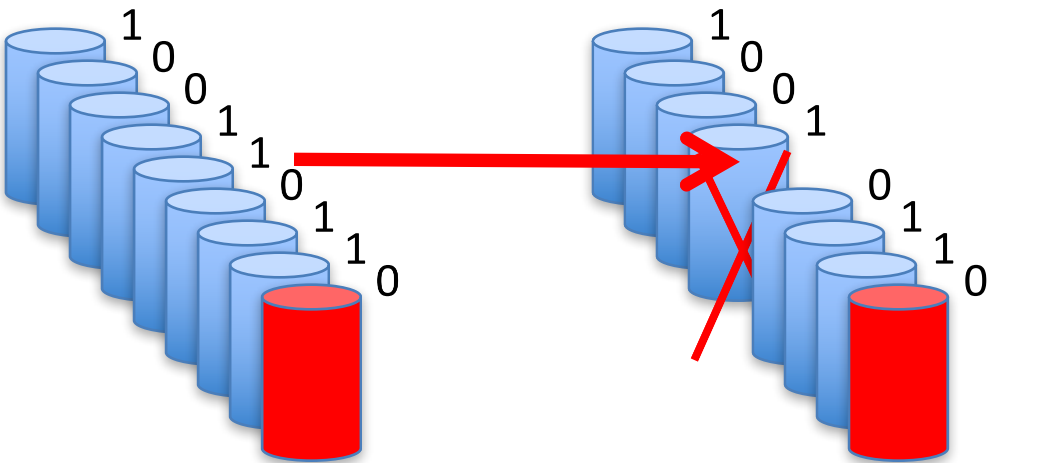
A) 8

B) $\log_2 8 = 3$

C) 1

D) 2

E) 0



Rule: Each bit row has an odd number of ones (odd parity)

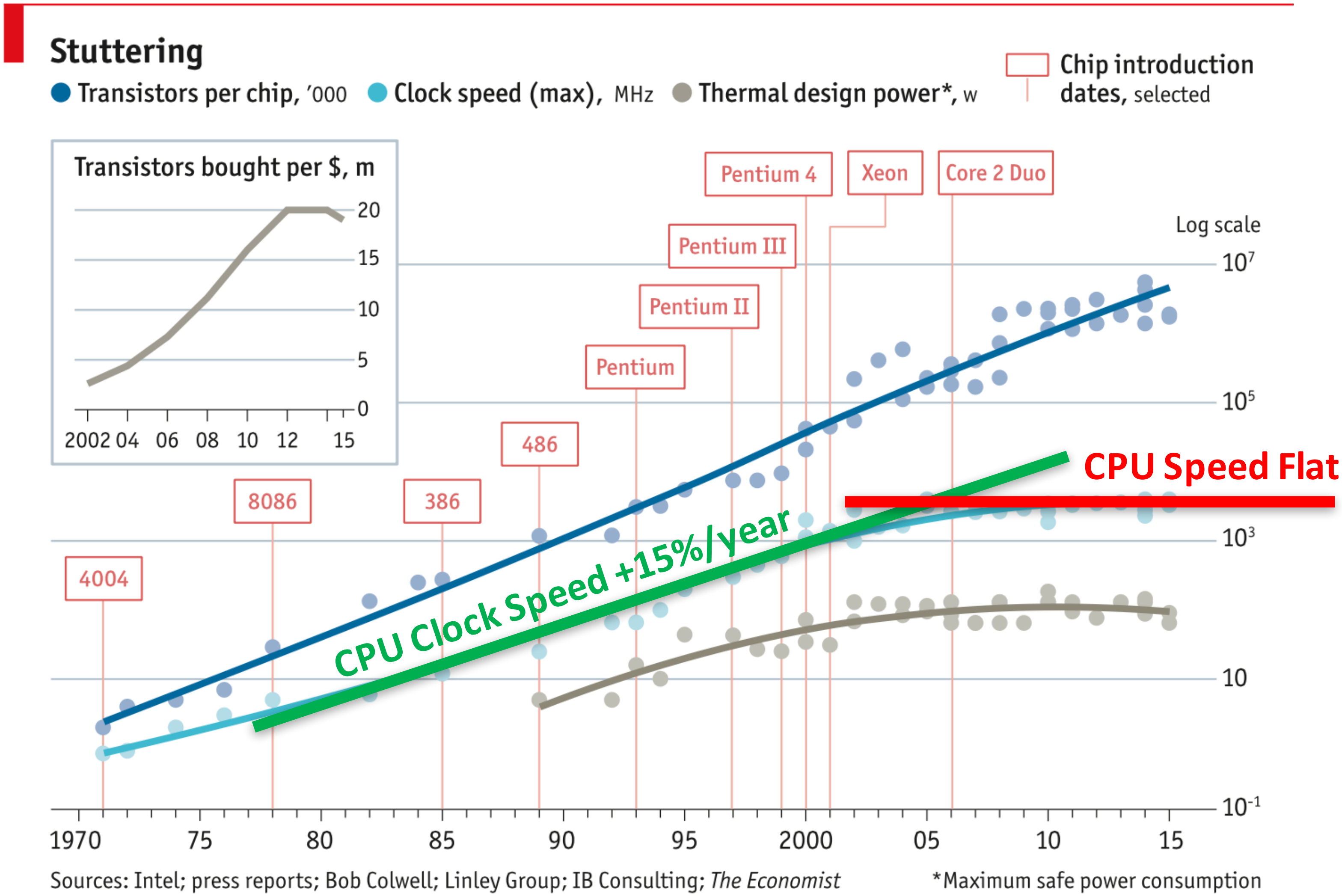
The missing bit must be one!

Break!

Understanding Computer Architecture

de.pinterest.com

Why is Architecture Exciting Today?



Motivation: Example

Code (C)

```
double sum(  // add up all values in array
        double *array,  // array
        int n,          // array size
        int inc)        // index increment
{
    double res = 0;
    int index = 0;
    for (int i = 0; i < n; i++) {
        index = (index + inc) % n;
        res += array[index];
    }
    return res;
}
```

Execution time

```
$ gcc array.c
$./a.out
1.0X time for inc        1
1.0X time for inc        3
2.3X time for inc 15485867
```

Consecutive is >2x faster!

Relevance:

- E.g. Vector vs HashMap
- Independent of programming language

Why?

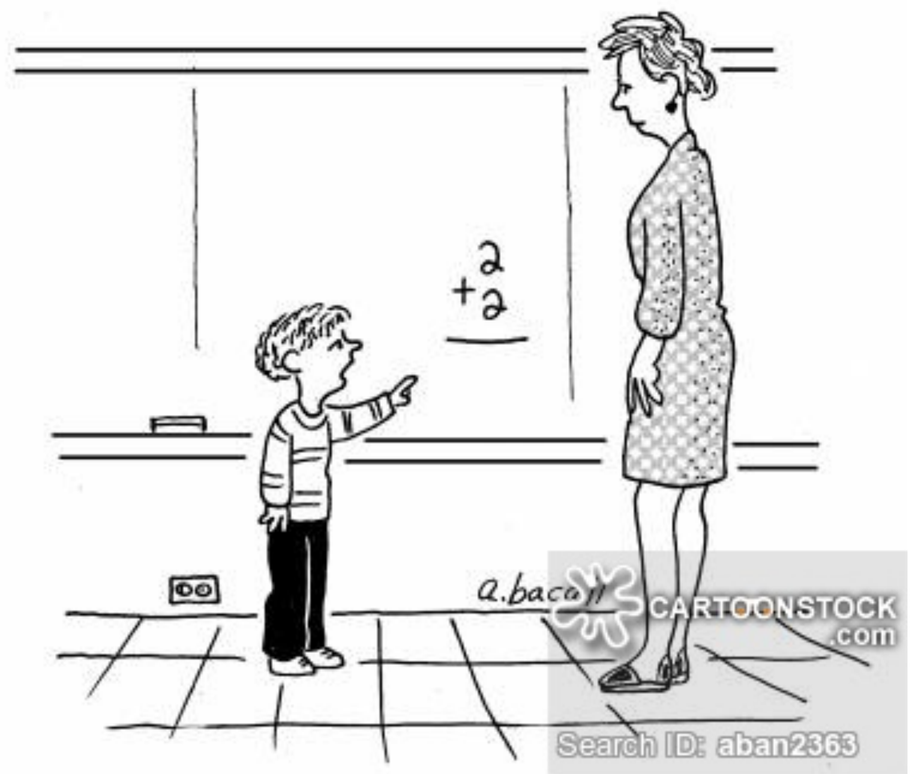
Complete Code, Page 1

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

/* sum elements in array of length n, use index increment inc
   note: for inc > 1, array elements are accessed non-consecutively
   beware: choose inc such that GCD(N, inc) = 0. N is the array size.
 */
double sum(  // add up all values in array
        double *array,  // array
        int n,          // array size
        int inc)        // index increment
{
    double res = 0;
    int index = 0;
    for (int i = 0; i < n; i++) {
        index = (index + inc) % n;
        res += array[index];
    }
    return res;
}

int main() {
    const int N = 50 * 1000 * 1000; // array size
    const int INC[] = { 1, 3, 15485867 };

    // create and initialize array
    double *array = malloc(N * sizeof (double));
    for (int i = 0; i < N; i++) array[i] = i;
    double element_sum = 0.5 * N * (N - 1);
```

Complete Code, Page 2

```
    // baseline ... processor time for increment = 1
    printf("Establish baseline ...\n");
    const int AVG = 10;
    double exec_for_inc_1 = 0;
    for (int i=0;  i<AVG;  i++) {
        clock_t start = clock();
        double s = sum(array, N, 1);
        exec_for_inc_1 += ((double)(clock()-start))/CLOCKS_PER_SEC;
    }
    exec_for_inc_1 /= AVG;

    // repeat to make sure result is consistent
    for (int r = 0; r < 2; r++) {
        for (int i = 0;  i <sizeof(INC)/sizeof(int); i++) {
            // clock measures processor time in units CLOCKS_PER_SEC
            clock_t start = clock();
            double s = sum(array, N, INC[i]);
            double exec_time = ((double)(clock()-start))/CLOCKS_PER_SEC;
            // check if we are getting the correct sum ...
            if (s != element_sum) printf("INCORRECT RESULT: %g != %g ",
                    s, element_sum);
            printf("%4.1f X time for increment %9d\n",
                    exec_time/exec_for_inc_1, INC[i]);
        }
        printf("\n");
    }
}
```

How to get an A in 61C



"Rather than learning how to solve that, shouldn't we be learning how to operate software that can solve that problem?"

Agenda

- Thinking about Machine Structures
- Great Ideas in Computer Architecture
- What You Need to Know About This Class
- Everything is a Number

Course Information

- Course Web: http://inst.eecs.Berkeley.edu/~cs61c/
- Instructors: Bernhard Boser and Randy H. Katz
- Teaching Staff:
  - Co-Head TAs: Derek Ahmed and Stephan Liu
  - Lead TAs: Manu Goyal, Rebecca Herman, Alex Khodaverdian, Jason Zhang
  - 20 TAs + 10 Tutors (see webpage)
- Textbooks: Average 15 pages of reading/week (can rent!)
  - Patterson & Hennessey, *Computer Organization and Design*, 5/e
  - Kernighan & Ritchie, *The C Programming Language*, 2nd Edition
  - Barroso & Holzle, *The Datacenter as a Computer, 2nd Edition*
- Piazza:
  - Every announcement, discussion, clarification happens there!

CS61c House Rules in a Nutshell

- Please don't disturb the instructors setting up or tearing down just before/after lecture
- Webcast? Yes!
- Labs and discussion after NEXT Tuesday's lecture
- Wait listed? Enroll in any available section/lab, swap later
- Excused Absences: Let us know by second week
- Midterms are in class **9/27** and **11/1;**
  Final is **12/16** at 7-10 PM
- Labs are partnered and Projects are solo (5)
  - Discussion is Good, but Co-Developing/Sharing/Borrowing Project Code or Circuits is Bad
  - No Public Repos Please: Don't Look, Don't Publish
- Join Piazza for more details … see http://inst.eecs.berkeley.edu/~cs61c/fa16/

Course Grading

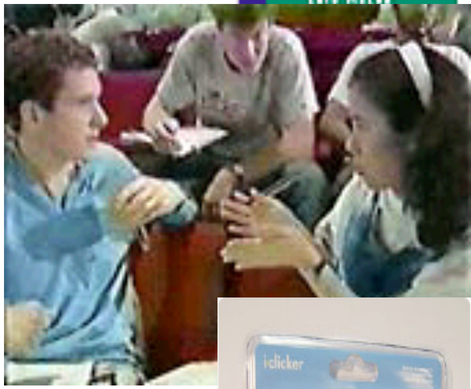
- EPA: Effort, Participation and Altruism (5%)
- Homework (5%)
- Partnered Labs (10%)
- Solo Projects (30%)
    1. Build your own Git repo (C)
    2. Non-Parallel Application (MIPS & C)
    3. Computer Processor Design (Logisim)
    4. Parallelize for Performance, SIMD, MIMD
    5. Massive Data Parallelism (Spark on Amazon EC2)
- Two midterms (12.5% each): **9/17** and **11/1**
- Final (25%): **12/16** @ 7-10pm (Last Exam Slot!)
- Performance Competition for honor (and EPA)

EPA!

- Effort
  - Attending prof and TA office hours, completing all assignments, turning in HW, doing reading quizzes
- Participation
  - Attending lecture and voting using the clickers
  - Asking great questions in discussion and lecture and making it more interactive
- Altruism
  - Helping others in lab or on Piazza
- EPA! points have the potential to bump students up to the next grade level! (but actual EPA! scores are internal)

Peer Instruction

- Increase real-time learning in lecture, test understanding of concepts vs. details
- As complete a “segment” ask multiple-choice question
  - 1-2 minutes to decide yourself
  - 2 minutes in pairs/triples to reach consensus.
  - Teach others!
  - 2 minute discussion of answers, questions, clarifications
- You can get transmitters from the ASUC bookstore
  - We’ll start this next week
  - No web-based clickers, sorry!



Late Policy … Slip Days!

- Assignments due at 11:59:59 PM
- You have 3 slip day tokens (NOT hour or min)
- Every day your project or homework is late (even by a minute) we deduct a token
- After you've used up all tokens, it's 33% deducted per day.
  - No credit if more than 3 days late
  - Save your tokens for projects, worth more!!
- No need for sob stories, just use a slip day!

39

- What do you think the policy should be?

Policy on Assignments and Independent Work

- With the exception of laboratories that explicitly permit you to work with partners, ALL OTHER WORK IS TO BE YOUR OWN.
- You are encouraged to discuss your assignments with other students, and extra credit will be assigned to students who help others via EPA, but what you hand in MUST BE YOUR OWN WORK.
- It is NOT acceptable to copy solutions from other students.
- It is NOT acceptable to copy (or start your) solutions from the Web.
- It is NOT acceptable to use PUBLIC github archives (whether looking at OR giving your answers away). PLEASE PUT A PASSWORD ON IT!
- We have tools and methods, developed over many years, for detecting this. You WILL be caught, and the penalties WILL be severe.
- **At the minimum F in the course**, and a letter to your university record documenting the incidence of cheating.
- (We've caught people in recent semesters!)
- **Both Giver and Receiver are equally culpable and suffer equal penalties**

Collaboration in Black and White

- **Good Collaboration**
  - High level discussion and brainstorming, stopping short of code snippets
- Bad Collaboration
  - Sitting together and co-writing code, inspecting each other’s code, taking (or giving) code whether in exchange or wholesale copying
- *This should be obvious, but …*
  - Don’t hire someone to do your assignments
  - Don’t ask someone outside of the class (a parent, a student from a previous semester, sourceforge) to help you
  - Don’t use search engines to look for solutions on-line or in someone’s unprotected GitHub (and don’t put your course project solutions in unprotected GitHubs please!)
  - It is supposed to be your own work after all!

https://www.interviewcake.com/google-interview-questions

Practice Questions for the Google Interview

Google is known for having one of the *hardest* technical interviews. So it's not surprising that the coding interview questions we hear about being asked at Google are some of our hardest. Get ready to nail your SWE, SRE or SET interview!

The Two Egg Problem »

A building has 100 floors. One of the floors is the highest floor an egg can be dropped from without breaking. If an egg is dropped from above that floor… keep reading »

Second Largest Item in BST »

Write a function to find the 2nd largest element in a binary search tree. Our first thought might be to do an in-order traversal of the BST, but this would take $O(n)$ time and… keep reading »

The Cake Thief »

You are a renowned thief who has recently switched from stealing precious metals to stealing cakes because of the insane profit margins. You want to make off with the most valuable haul possible, and you… keep reading »

What Characterizes A Google Interview Question?

What makes a Google interview question different from one that might be asked at Facebook, Amazon, Microsoft, Twitter, etc?

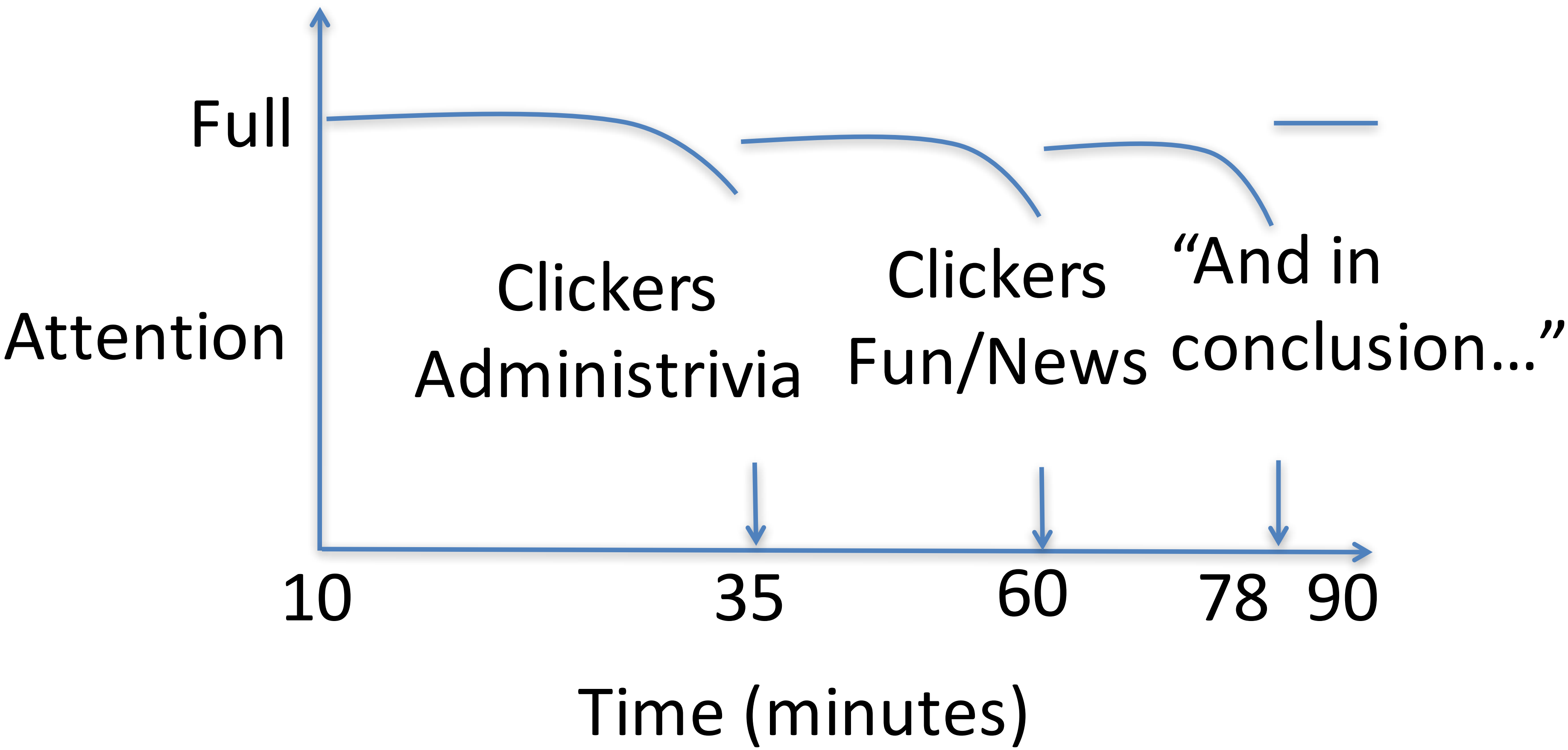
Nothing. Nothing at all.

The truth is, the specific question you get asked has far more to do with the *interviewer* assigned to you than it does the *company* you're interviewing at.

There's no way to know ahead of time what questions your interviewers will ask you. Your interviewers' *employer* probably doesn't even know what questions your interviewers will ask you. There are literally thousands of possibilities for what your interviewer could ask you. So the strategy for winning at these interviews is *not* to "learn" a bunch of Google interview questions and then hope that your interviewers ask you the questions you've already learned

Send me a message!

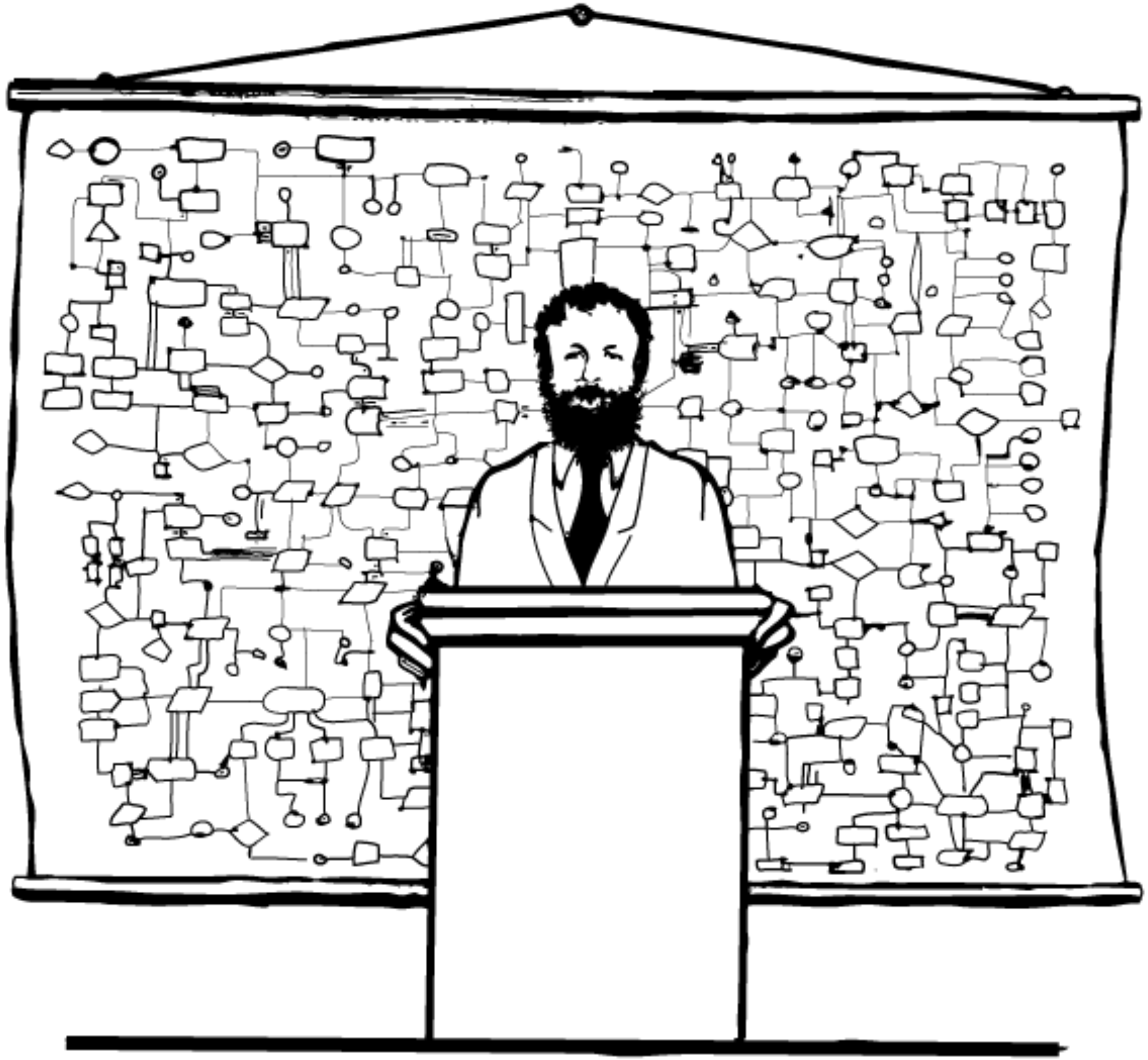
Architecture of a typical Lecture



Break!

Agenda

- Thinking about Machine Structures
- Great Ideas in Computer Architecture
- What You Need to Know About This Class
- Everything is a Number



"Now that you have an overview of the system, we're ready for a little more detail"

http://www.ruthmalan.com

Computer Data

- Computers represent data as binary values
- Unit element: *bit*
  - Just two possible values, 0 or 1
  - Can be efficiently stored/communicated/manipulated in hardware
- Use many bits to store more complex information, e.g.
  - Byte: 8 bits, can represent 2^8 = 256 different values
  - Word, e.g. 4 bytes (32 bits) to represent 2^{32} different values
  - 64-bit floating point numbers
  - Text files, databases, … (many bytes)
  - *Computer program*

Binary Number Conversion

Binary → Decimal

$1001010_{two} = ?_{ten}$

Binary Digit	Decimal Value
0	$0 \times 2^0 = 0$
1	$1 \times 2^1 = 2$
0	$0 \times 2^2 = 0$
1	$0 \times 2^3 = 8$
0	$0 \times 2^4 = 0$
0	$0 \times 2^5 = 0$
1	$1 \times 2^6 = 64$
	$\Sigma = 74_{ten}$

Decimal → Binary

$74_{ten} = ?two$

Decimal	Binary (odd?)
74	0
/2 = 37	1
/2 = 18	0
/2 = 9	1
/2 = 4	0
/2 = 2	0
/2 = 1	1
Collect →	1001010_{two}

Hexadecimal

- Problem: many digits
  - e.g. $7643_{ten} = 1110111011011_{two}$
- Solutions:
  - Grouping: $1\ 1101\ 1101\ 1011_{two}$
  - Hexadecimal: $1DDB_{hex}$
  - Octal: $1\ 110\ 111\ 011\ 011_{two}$

    16733_{oct}

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

The Computer Knows it, too

```c
#include <stdio.h>

int main() {
    const int N = 1234;

    printf("Decimal: %d\n", N);
    printf("Hex:     %x\n", N);
    printf("Octal:   %o\n", N);

    printf("Literals (not supported by all compilers):\n");
    printf("0x4d2          = %d (hex)\n", 0x4d2);
    printf("0b10011010010 = %d (binary)\n", 0b10011010010);
    printf("02322          = %d (octal, prefix 0 - zero)\n", 02322);
}
```

Output

```
Decimal: 1234
Hex:     4d2
Octal:   2322
Literals (not supported by all compilers):
0x4d2         = 1234 (hex)
0b10011010010 = 1234 (binary)
02322         = 1234 (octal, prefix 0 - zero)
```

Large Numbers

Decimal

Suffix	Multiplier	Value
K	10^3	1000
M	10^6	1000,000
G	10^9	1000,000,000
T	10^{12}	1000,000,000,000

Binary (IEC)

Suffix	Multiplier	Value
Ki	2^{10}	1024
Mi	2^{20}	1048,576
Gi	2^{30}	1073,741,824
Ti	2^{40}	1099,511,627,776

E.g. 1GiByte disk versus 1GByte disk

Marketing exploits this: 1TB disk → 100GB less than 1TiB

https://en.wikipedia.org/wiki/Byte

Signed Integer Representation

Sign & magnitude (8-bit example):

sign	7-bit magnitude (0 … 127)

Rules for addition, a + b:

- If (*a>0 and b>0):* add, sign 0
- If (*a>0 and b<0):* subtract, sign …
- …
- +0, -0 → are they equal? comparator must handle special case!

Cumbersome

- ”Complicated” hardware: reduced speed / increased power
- ***Is there a better way?***

4-bit Example

Decimal			Binary	
7	7	7	0111	
+ -3	+ -3 + 16	+ 13	+ 1101	
4	4 + 16	4 + 16	1 0100	0100 + 1 0000

- Map negative → positive numbers
  - Example for N=4-bit: -3 → $2^4 - 3 = 13$
  - "Two's complement"
  - No special rules for adding positive and negative numbers

-8	-7	…	-1	0	1	…	7
				0	1	…	7
				8	9	…	15

$+ 2^4 = 16$

Two's Complement

(8-bit example)

Signed Decimal	Unsigned Decimal	Binary Two's Complement
-128	128	1 0 0 0 0 0 0 0
-127	129	1 0 0 0 0 0 0 1
…	…	…
-2	254	1 1 1 1 1 1 1 0
-1	255	1 1 1 1 1 1 1 1
0	0	0 0 0 0 0 0 0 0
1	1	0 0 0 0 0 0 0 1
…	…	. . .
127	127	0 1 1 1 1 1 1 1

+256 (−128 … −1 → 128 … 255)

+0 (0 … 127 → 0 … 127)

Note: Most significant bit (MSB) equals sign

Unary Negation (Two's Complement)

4-bit Example (-8_{ten} ... $+7_{ten}$)

Brute Force & Tedious

"largest" 4-bit number + 1

$$\begin{array}{rr} 16_{ten} & 10000_{two} \\ -\ 3_{ten} & -\ 0011_{two} \\ \hline 13_{ten} & 1101_{two} \end{array}$$

$$\begin{array}{rr} 16_{ten} & 10000_{two} \\ -\ 13_{ten} & -\ 1101_{two} \\ \hline 3_{ten} & 0011_{two} \end{array}$$

Clever & Elegant

$$\begin{array}{rr} 15_{ten} & 01111_{two} \\ -\ 3_{ten} & -\ 0011_{two} \\ \hline 12_{ten} & 1100_{two} \\ +\ 1_{ten} & +\ 0001_{two} \\ \hline 13_{ten} & 1101_{two} \end{array}$$

invert

Your Turn

- What is the decimal value of the following binary 8-bit 2's complement number?

$\mathbf{1110\ 0001}_{two}$

Answer	Value
A	33_{ten}
B	-31_{ten}
C	225_{ten}
D	-33_{ten}
E	None of the above

Addition

4-bit Example

Unsigned

$$\begin{array}{rr} & 3_{ten} \\ + & 4_{ten} \\ \hline & 7_{ten} \end{array} \qquad \begin{array}{rr} & 0011_{two} \\ + & 0100_{two} \\ \hline & 0111_{two} \end{array}$$

$$\begin{array}{rr} & 3_{ten} \\ + & 11_{ten} \\ \hline & 14_{ten} \end{array} \qquad \begin{array}{rr} & 0011_{two} \\ + & 1011_{two} \\ \hline & 1110_{two} \end{array}$$

Signed (Two's Complement)

$$\begin{array}{rr} & 3_{ten} \\ + & 4_{ten} \\ \hline & 7_{ten} \end{array} \qquad \begin{array}{rr} & 0011_{two} \\ + & 0100_{two} \\ \hline & 0111_{two} \end{array}$$

$$\begin{array}{rr} & 3_{ten} \\ + & -5_{ten} \\ \hline & -2_{ten} \end{array} \qquad \begin{array}{rr} & 0011_{two} \\ + & 1011_{two} \\ \hline & 1110_{two} \end{array}$$

No special rules for two's complement signed addition

Overflow

4-bit Example

Unsigned

$$\begin{array}{r} 13_{ten} \\ +\ 14_{ten} \\ \hline 27_{ten} \end{array} \qquad \begin{array}{r} 1101_{two} \\ +\ 1110_{two} \\ \hline (1)1011_{two} \end{array}$$

carry-out and overflow

$$\begin{array}{r} 7_{ten} \\ +\ 1_{ten} \\ \hline 8_{ten} \end{array} \qquad \begin{array}{r} 0111_{two} \\ +\ 0001_{two} \\ \hline (0)1000_{two} \end{array}$$

no carry-out and no overflow

Carry-out → Overflow

Signed (Two's Complement)

$$\begin{array}{r} -3_{ten} \\ +\ -2_{ten} \\ \hline -5_{ten} \end{array} \qquad \begin{array}{r} 1101_{two} \\ +\ 1110_{two} \\ \hline (1)1011_{two} \end{array}$$

carry-out but no overflow

$$\begin{array}{r} 7_{ten} \\ +\ 1_{ten} \\ \hline -8_{ten} \end{array} \qquad \begin{array}{r} 0111_{two} \\ +\ 0001_{two} \\ \hline (0)1000_{two} \end{array}$$

no carry-out but overflow

Carry-out ↛ Overflow

Overflow Detection

4-bit Example

Unsigned

- Carry-out indicates overflow

Signed (Two's Complement)

- Overflow if
  - Signs of operands are equal

    AND
  - Sign of result differs from sign of operands
- No overflow when signs of operands differ

Overflow rules depend on operands (signed vs unsigned)

Sign Extension

Decimal	Binary		
	4-bit	8-bit	32-bit
3_{ten}	0011_{two}	$0000\ 0011_{two}$	$0000\ 0000\ 0000\ 0011_{two}$
-3_{ten}	1101_{two}	$1111\ 1101_{two}$	$1111\ 1111\ 1111\ 1101_{two}$

- Why is this relevant?
- Assignment differs for signed (above) and unsigned numbers
  - Compiler knows (from type declaration)
  - Different assembly instructions for copying signed/unsigned data

Your Turn

- Which range of decimals can be expressed with a 6-bit two's complement number?

Answer	Range
A	-32 … 32
B	-64 … 63
C	-31 … 32
D	-16 … 15
E	-32 … 31

Answer

- Which range of decimals can be expressed with a 6-bit two's complement number?

Answer	Range
A	-32 ... 32
B	-64 ... 63
C	-31 ... 32
D	-16 ... 15
E	-32 ... 31

And In Conclusion … (1/2)

- CS61C:
  - High performance by leveraging computer architecture:
    - Strength and weaknesses (e.g. cache)
    - Performance features (e.g. parallel instructions)
  - Learn C and assembly facilitate access to machine features
- Basis: five great ideas in computer architecture
  1. Abstraction: Layers of Representation/Interpretation
  2. Moore's Law
  3. Principle of Locality/Memory Hierarchy
  4. Parallelism
  5. Dependability via Redundancy
- Performance Measurement and Improvement

And In Conclusion … (2/2)

- Everything is a Number!
  - Collections of bits can store and communicate arbitrary digital data
  - Even programs are represented by bits
- Two's complement representation avoids special rules for addition of negative numbers

65

And in Conclusion: Everything is a Number

https://www.cartoonstock.com