

Machine Learning's Slides

Author: Pannenets.F

Date: September 16, 2020

Je reviendrai et je serai des millions. «Spartacus»

Part I

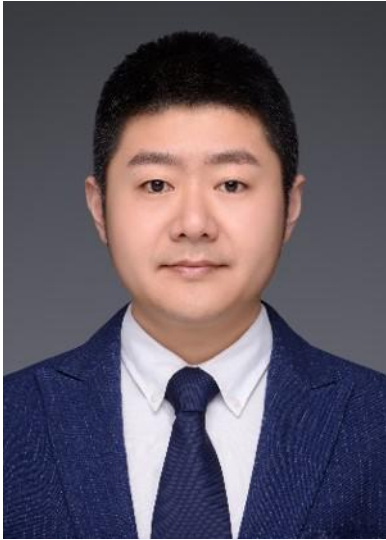
part

Chapter 1 Week1: Intro

1.1 Introduction

Computer Architecture (计算机体系结构)

Lecture #1 – Introduction



Lecturer Yuanqing Cheng (成元庆)

School of Microelectronics

Beihang University

www.cadetlab.cn

“I stand on the shoulders of giants...”



**Prof
David
Patterson**



**Prof
Dan
Garcia**

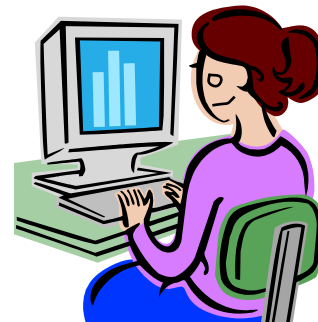
**Thanks to these talented folks (& many others)
whose contributions have helped make this
course a really tremendous course!**

What are prerequisites of this course ?

- **Programming languages ?**
- **Data structures ?**
- **Digital logic design ?**

Are Computers Smart?

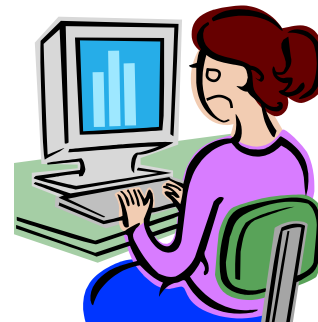
- To a programmer:
 - Very complex operations / functions:
 - `(map (lambda (x) (* x x)) '(1 2 3 4))`
 - Automatic memory management:
 - `List l = new List;`
 - “Basic” structures:
 - Integers, floats, characters, plus, minus, print commands



Computers
are smart!

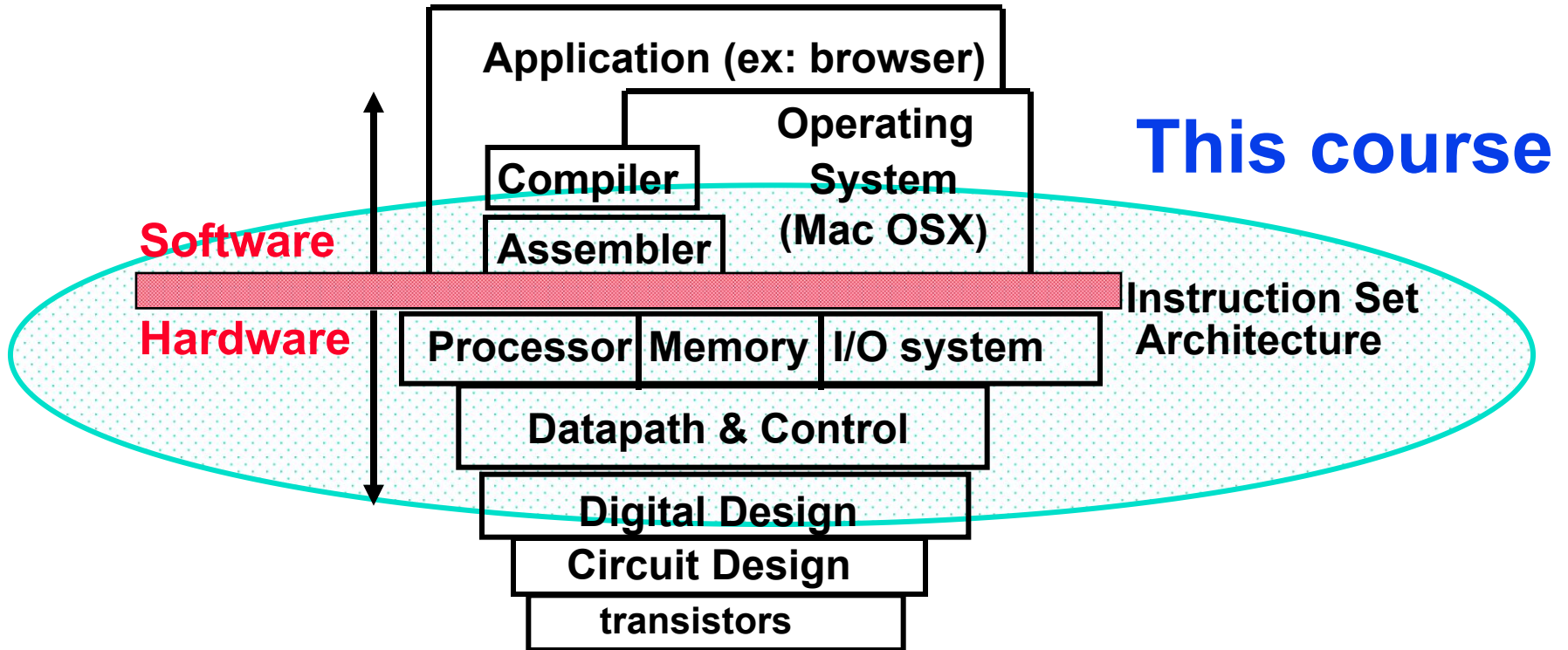
Are Computers Smart?

- In real life at the lowest level:
 - Only a handful of operations:
 - {and, or, not}
 - No automatic memory management.
 - Only 2 values:
 - {0, 1} or {low, high} or {off, on}



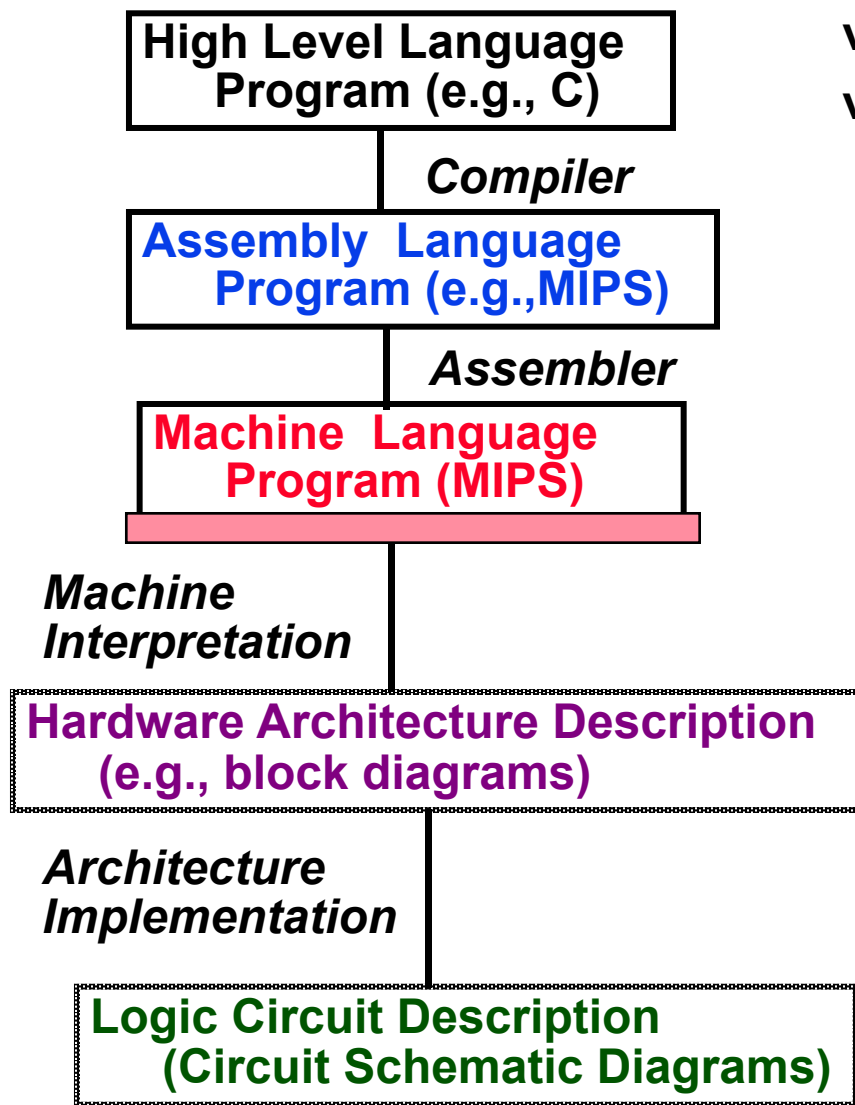
**Computers
are dumb!**

What are “Computer Architecture”?



**Coordination of many
*levels (layers) of abstraction***

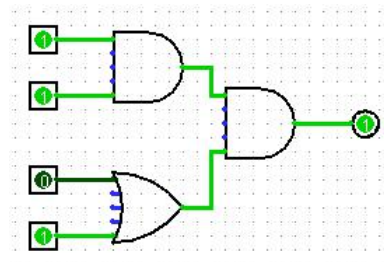
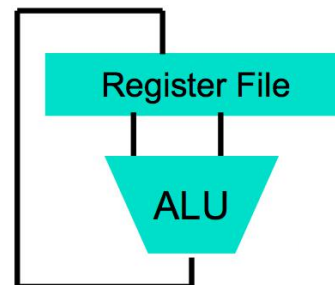
Levels of Representation



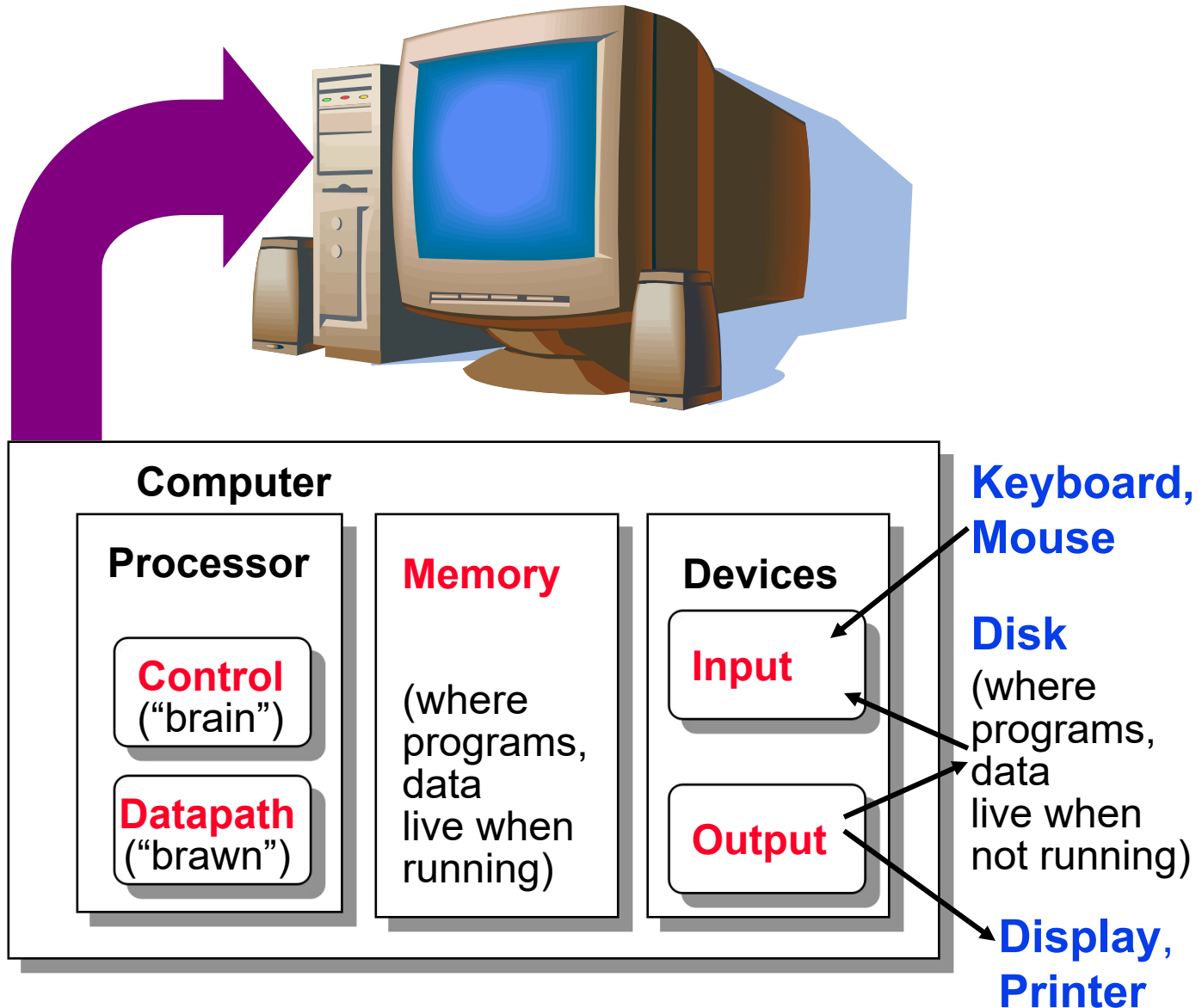
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

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

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



Anatomy: 5 components of any Computer



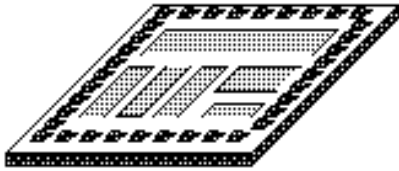
Overview of Physical Implementations

The hardware out of which we make systems.

- **Integrated Circuits (ICs)**
 - **Combinational logic circuits, memory elements, analog interfaces.**
- **Printed Circuits (PC) boards**
 - **substrate for ICs and interconnection, distribution of CLK, Vdd, and GND signals, heat dissipation.**
- **Power Supplies**
 - **Converts line AC voltage to regulated DC low voltage levels.**
- **Chassis (rack, card case, ...)**
 - **holds boards, power supply, provides physical interface to user or other systems.**
- **Connectors and Cables.**

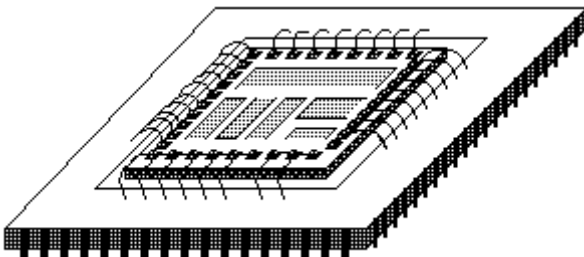
Integrated Circuits (2020 state-of-the-art)

Bare Die



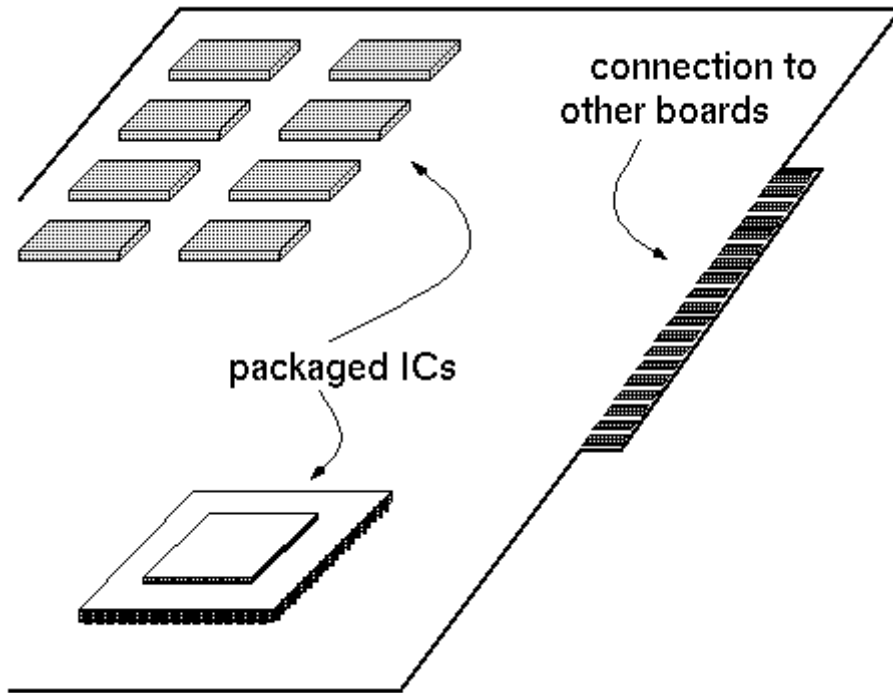
- Primarily Crystalline Silicon
- 1mm - 25mm on a side
- feature size ~ 14/7 nm
- Billions of transistors
- (25 - 100M “logic gates”)
- 3 - 12 conductive layers
- “CMOS” (complementary metal oxide semiconductor) - most common.

Chip in Package



- Package provides:
 - spreading of chip-level signal paths to board-level
 - heat dissipation.
- Ceramic or plastic with gold wires.

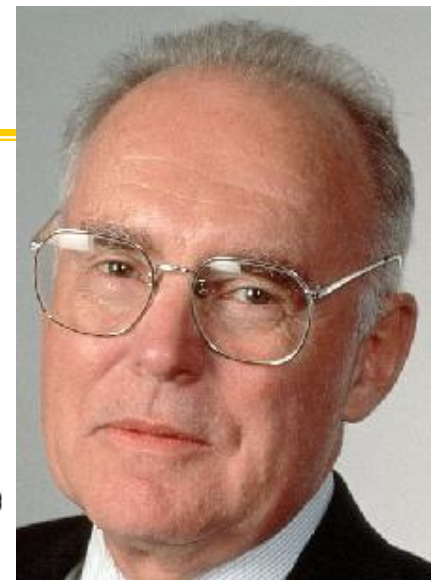
Printed Circuit Boards



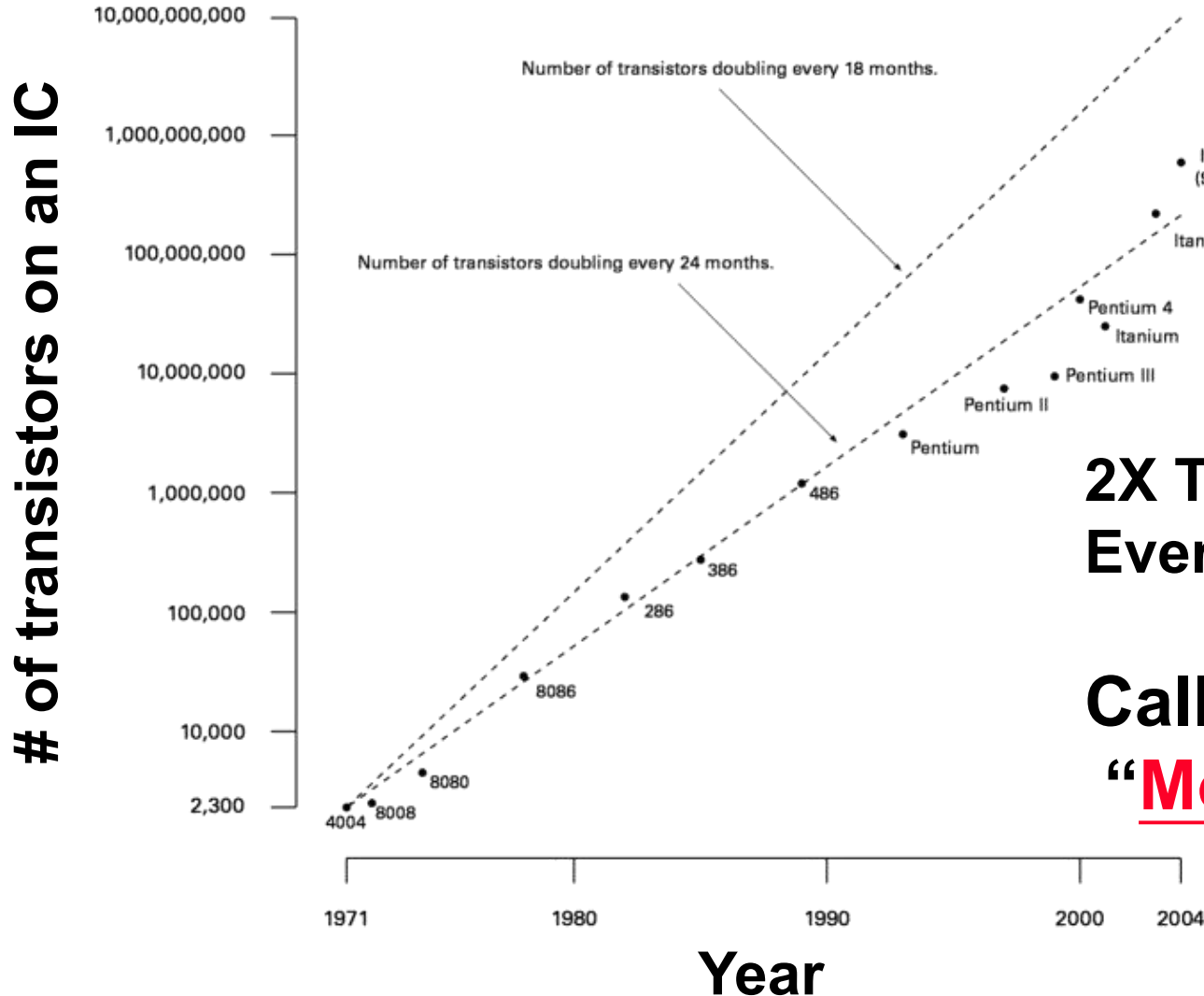
- **fiberglass or ceramic**
- **1-20 conductive layers**
- **1-20 in on a side**
- **IC packages are soldered down.**
- **Provides:**
 - **Mechanical support**
 - **Distribution of power and heat.**

Technology Trends:

Microprocessor Complexity



Gordon Moore
Intel Cofounder!



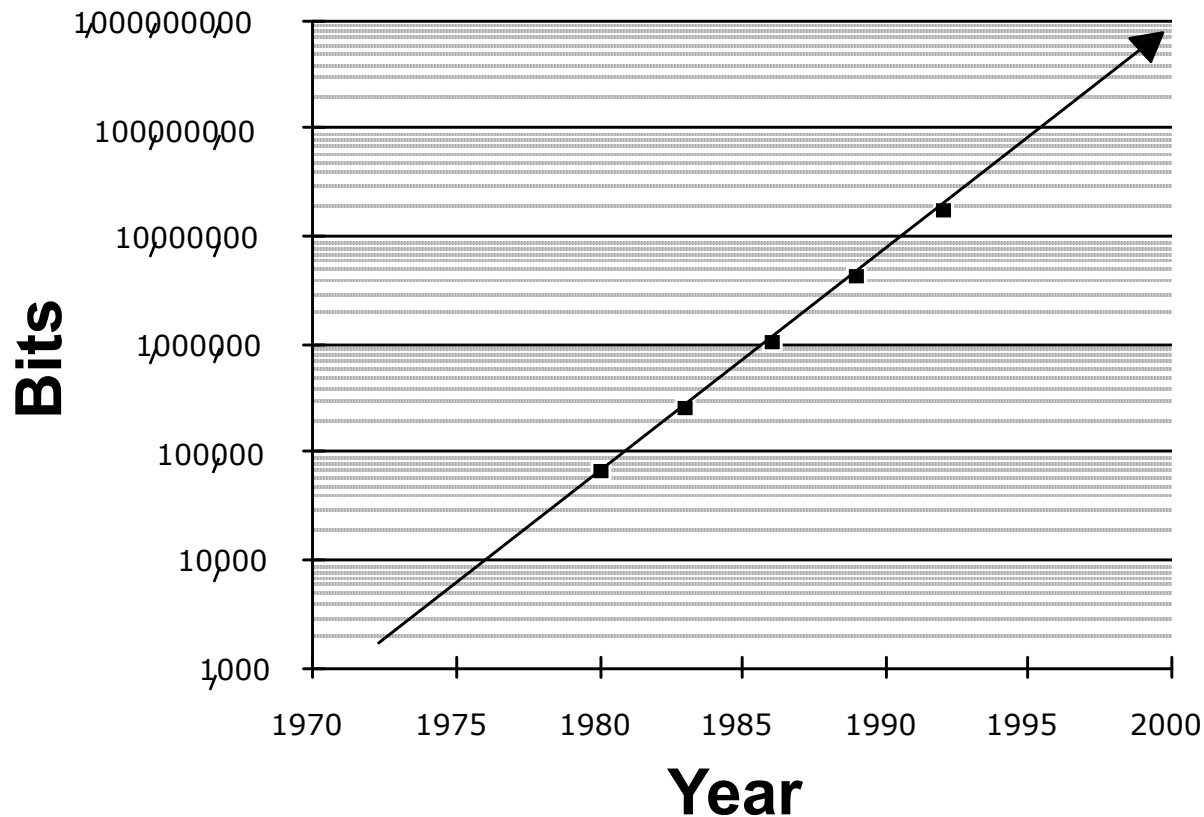
2X Transistors / Chip
Every 1.5 years

Called
“Moore’s Law”

Technology Trends: Memory Capacity

(Single-Chip DRAM)

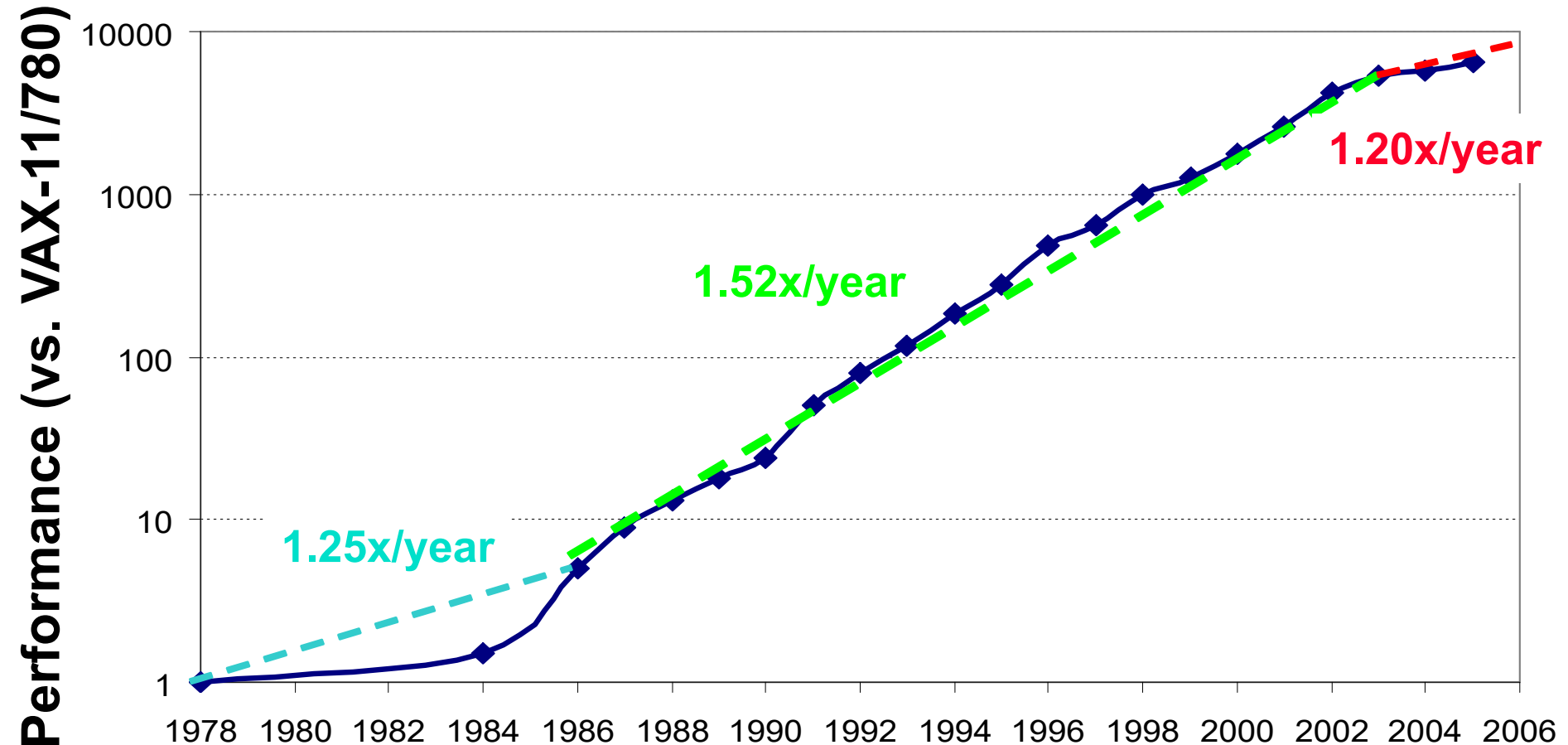
size



- **Now 1.4X/yr, or 2X every 2 years.**
- **8000X since 1980!**

| year | size (Mbit) |
|------|--------------|
| 1980 | 0.0625 |
| 1983 | 0.25 |
| 1986 | 1 |
| 1989 | 4 |
| 1992 | 16 |
| 1996 | 64 |
| 1998 | 128 |
| 2000 | 256 |
| 2002 | 512 |
| 2004 | 1024 |
| | (1Gbit) |
| 2006 | 2048 (2Gbit) |
| 2010 | 8192 (8Gbit) |

Technology Trends: Uniprocessor Performance (SPECint)



- VAX : 1.25x/year 1978 to 1986
- RISC + x86: 1.52x/year 1986 to 2002
- RISC + x86: 1.20x/year 2002 to present

Computer Technology - Dramatic Change!

- **Memory**

- DRAM capacity: 2x / 2 years (since '96);
64x size improvement in last decade.

- **Processor**

- Speed 2x / 1.5 years (since '85); **[slowing!]**
Now almost remain the same.

- **Disk**

- Capacity: 2x / 1 year (since '97)
250X size in last decade.

Computer Technology - Dramatic Change!

You just learned the difference between (Kilo, Mega, ...) and (Kibi, Mebi, ...)!

- **State-of-the-art PC :**
(at least...)

- **Processor clock speed:** 4,000 **Mega**Hertz
(4.0 **Giga**Hertz)
- **Memory capacity:** 65,536 **Mebi**Bytes
(64.0 **Gibi**Bytes)
- **Disk capacity:** 2,000 **Giga**Bytes
(2.0 **Tera**Bytes)
- **New units!** **Mega** \Rightarrow **Giga**, **Giga** \Rightarrow **Tera**

(**Tera** \Rightarrow **Peta**, **Peta** \Rightarrow **Exa**, **Exa** \Rightarrow **Zetta**
Zetta \Rightarrow **Yotta** = 10^{24})

Computer arch. : So, what's in it for me?

- **Learn some of the big ideas in CS & Engineering:**
 - **Principle of abstraction**
 - Used to build systems as layers
 - **5 Classic components of a Computer**
 - **Data can be anything**
 - Integers, floating point, characters, ...
 - A program determines what it is
 - Stored program concept: instructions just data
 - **Principle of Locality**
 - Exploited via a memory hierarchy (cache)
 - **Greater performance by exploiting parallelism**
 - **Compilation v. interpretation through system layers**
 - **Principles / Pitfalls of Performance Measurement**

Others Skills learned in this course

- **Enhance C programming skill**
 - If you know one, you should be able to learn another programming language largely on your own
 - If you know C++ or Java, it should be easy to pick up their ancestor, C
- **Assembly Language Programming**
 - This is a skill you will pick up, as a side effect of understanding the Big Ideas
- **Hardware design**
 - We'll learn just the basics of hardware design
 - We'll this in more detail in following courses

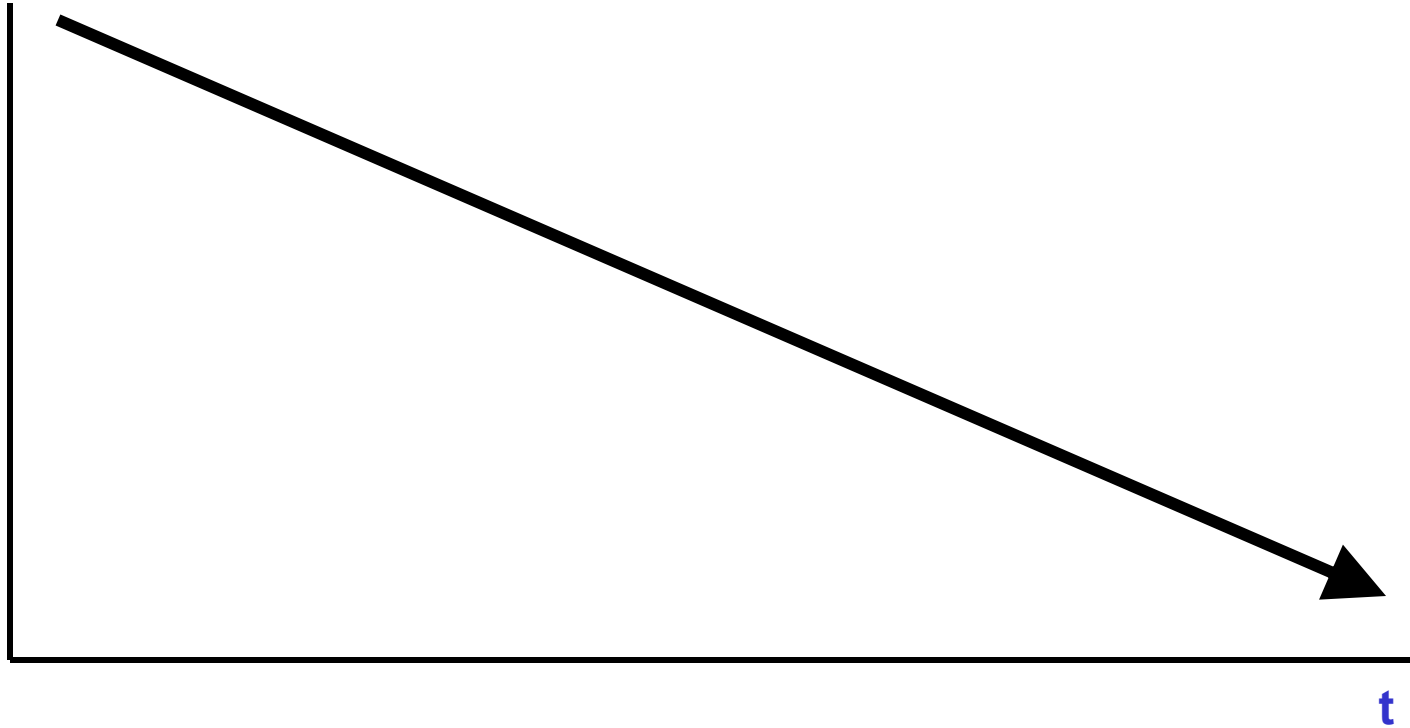
Yoda says...

**“Always in
motion is the
future...”**



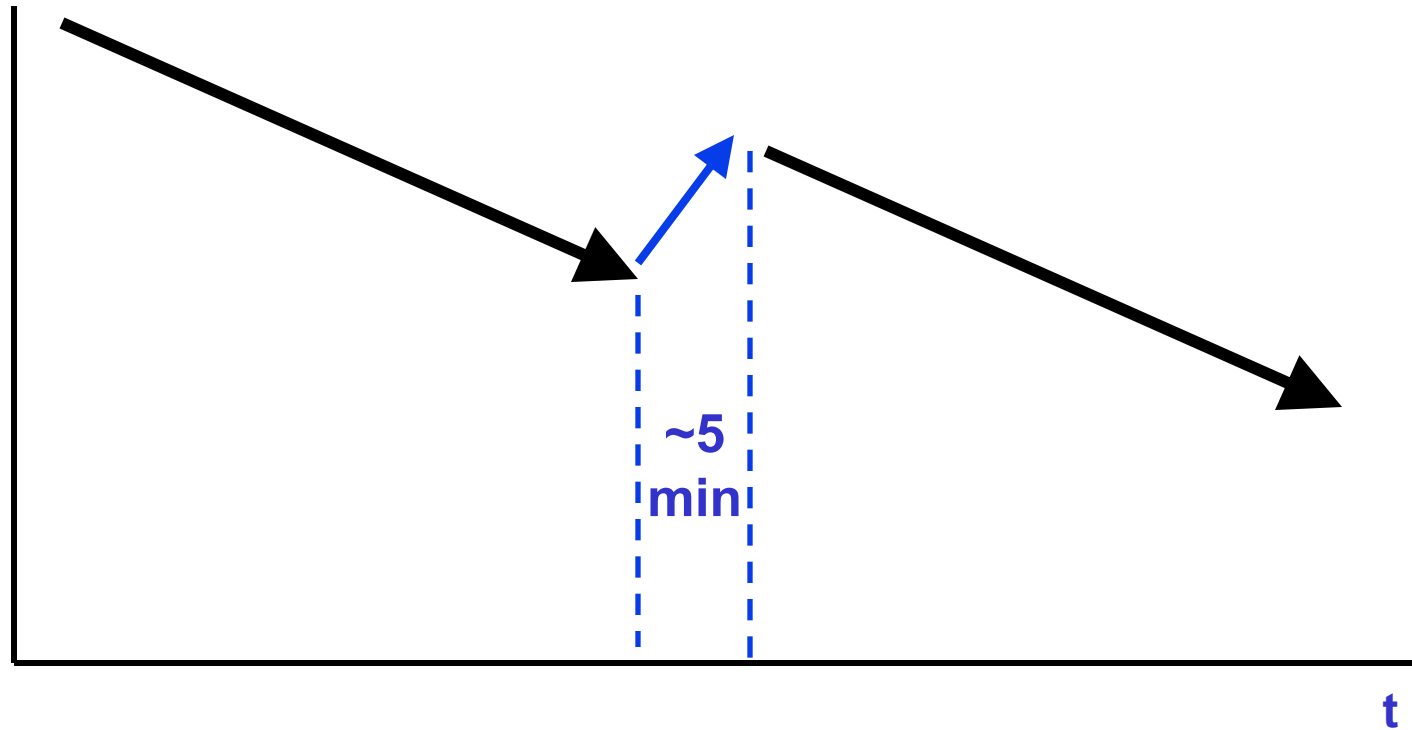
**Our schedule may change slightly depending on some factors.
This includes lectures, assignments & labs...**

What is this?



Attention over time!

What is this?!



Attention over time!

Tried-and-True Technique: 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
 - 3 minutes in pairs/triples to reach consensus. Teach others!
 - 5-7 minute discussion of answers, questions, clarifications



Extra Credit: EPA!

- **Effort**
 - Attending my or my TA's office hours, completing all assignments, turning in HW, doing reading quizzes
- **Participation**
 - Attending lecture and voting in Peer Instruction
 - Asking great questions in discussion and lecture and making it more interactive
- **Altruism**
 - Helping others in lab or wechat
- **EPA! extra credit points have the potential to bump students up to the next grade level!**

Course Problems...Cheating

- What is cheating?
 - Studying together in groups is encouraged.
 - Turned-in work must be completely your own.
 - Common examples of cheating: running out of time on a assignment and then pick up output, take homework from box and copy, person asks to borrow solution “just to take a look”, copying an exam question, ...
 - You’re not allowed to work on homework/projects/exams with anyone (other than ask Qs walking out of lecture)
 - Both “giver” and “receiver” are equally culpable
- Cheating points: **0 EPA, negative points for that assignment / project / exam** (e.g., if it’s worth 10 pts, you get -10) **In most cases, F in the course. .**

My goal as an instructor

- **To make your experience in this course as enjoyable & informative as possible**
 - **Enthusiasm, graphics & technology-in-the-news in lecture**
 - **Fun, challenging projects & HW**
 - **Pro-student policies**
- **To be a good-teaching man**
 - **Please give me feedback so I improve!**
Why am I not excellent teacher for you? I will listen!!



Teaching Assistants

- **Jiacheng Ni**



Summary

- **Continued rapid improvement in computing**
 - **2X every 2.0 years in memory size;**
every 1.5 years in processor speed;
every 1.0 year in disk capacity;
 - **Moore's Law enables processor**
(2X transistors/chip ~1.5-2 yrs)
- **5 classic components of all computers**
Control Datapath Memory Input Output



Processor

Reference slides

You ARE responsible for the material on these slides (they're just taken from the reading anyway) ; we've moved them to the end and off-stage to give more breathing room to lecture!

Course Lecture Outline

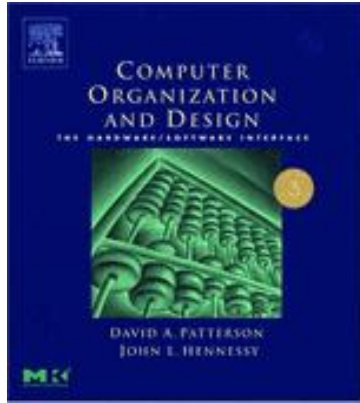
- **Machine Representations**
 - Numbers (integers, reals)
 - Assembly Programming
 - Compilation, Assembly
- **Processors & Hardware**
 - Logic Circuit Design
 - CPU organization
 - Pipelining
- **Memory Organization**
 - Caches
 - Virtual Memory
- **I / O**
 - Interrupts
 - Disks, Networks
- **Advanced Topics**
 - Performance
 - Virtualization
 - Parallel Programming

Homeworks and Projects

- Homework exercises
- Projects
- All exercises, reading, homeworks, projects on course web page
www.cadetlab.cn/courses
- We will DROP your lowest HW, Lab!
- Only one final exam

Your final grade

- **Grading (max: 100 pts)**
 - 20pts = 20% Homework
 - 20pts = 20% Projects
 - 50pts = 50% Final exam
 - 10pts = 10% Attendance
 - Extra EPA points (5pts)



- Required: *Computer Organization and Design: The Hardware/Software Interface, Third Edition*, Patterson and Hennessy (COD).
- Reading assignments on web page

Peer Instruction and Just-in-time-learning

- **Read textbook**
 - Reduces examples have to do in class
 - Get more from lecture (also good advice)
- **Fill out 3-question on web**
 - Graded for effort, not correctness...
 - This counts toward EPA score

1.2 Numbers

Computer Architecture (计算机体系结构)

Lecture #2 – Number Representation



Lecturer Yuanqing Cheng (成元庆)

www.cadetlab.cn



News & Analysis

Qualcomm Prepares to Take
Nvidia for a Ride

Review



- Continued rapid improvement in computing
 - 2X every 2.0 years in memory size;
every 1.5 years in processor speed;
every 1.0 year in disk capacity;
 - Moore's Law enables processor
(2X transistors/chip every 2 yrs)
- 5 classic components of all computers

a b c d e
Control Datapath Memory Input Output



Processor

**What'll be the most
important part of a computer
in the future?**

Putting it all in perspective...

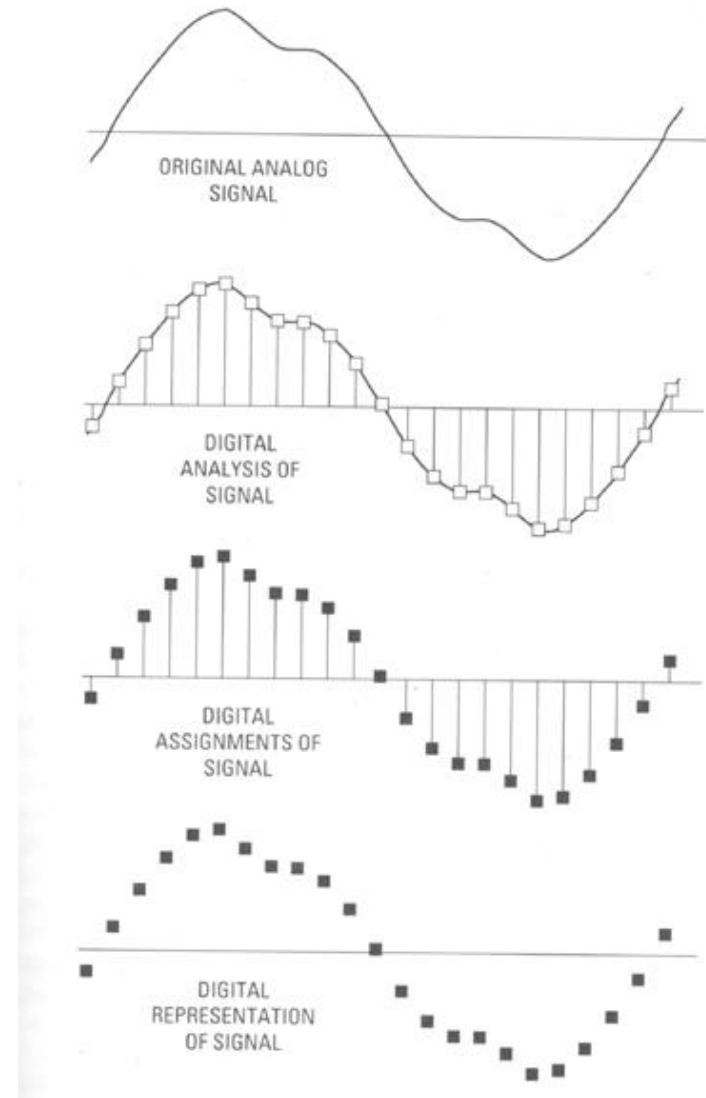
“If the automobile had followed the same development cycle as the computer,

– *Robert X. Cringely*



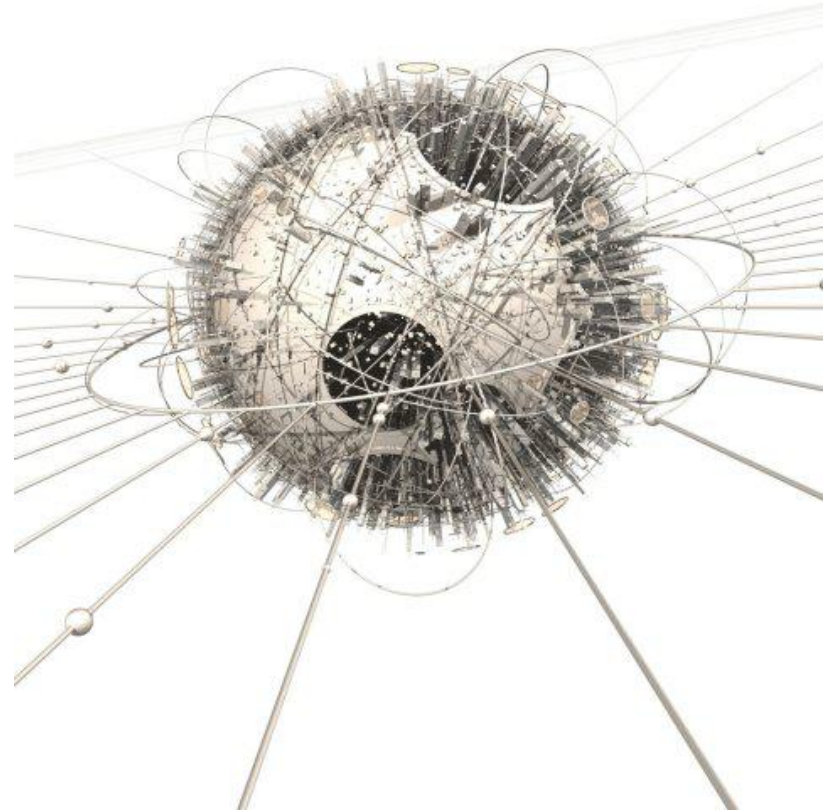
Data input: Analog → Digital

- Real world is analog!
- To import analog information, we must do two things
 - **Sample**
 - E.g., for a CD, every 44,100ths of a second, we ask a music signal how loud it is.
 - **Quantize**
 - For every one of these samples, we figure out where, on a 16-bit (65,536 tick-mark) “yardstick”, it lies.



www.joshuadysart.com/journal/archives/digital_sampling.gif

Digital data not nec born Analog...



hof.povray.org

BIG IDEA: Bits can represent anything!!

- Characters?

- 26 letters \Rightarrow 5 bits ($2^5 = 32$)
- upper/lower case + punctuation \Rightarrow 7 bits (in 8) (“ASCII”)
- standard code to cover all the world’s languages \Rightarrow 8,16,32 bits (“Unicode”) www.unicode.com



- Logical values?

- 0 \Rightarrow False, 1 \Rightarrow True

- colors ? Ex: Red (00) Green (01) Blue (11)

- locations / addresses? commands?

- **MEMORIZE:** N bits \Leftrightarrow at most 2^N things

How many bits to represent π ?



- a) 1
- b) 9 ($\pi = 3.14$, so that's 011 “.” 001 100)
- c) 64 (Since Macs are 64-bit machines)
- d) Every bit the machine has!
- e) ∞

What to do with representations of numbers?

- **Just what we do with numbers!**

- **Add them**
- **Subtract them**
- **Multiply them**
- **Divide them**
- **Compare them**

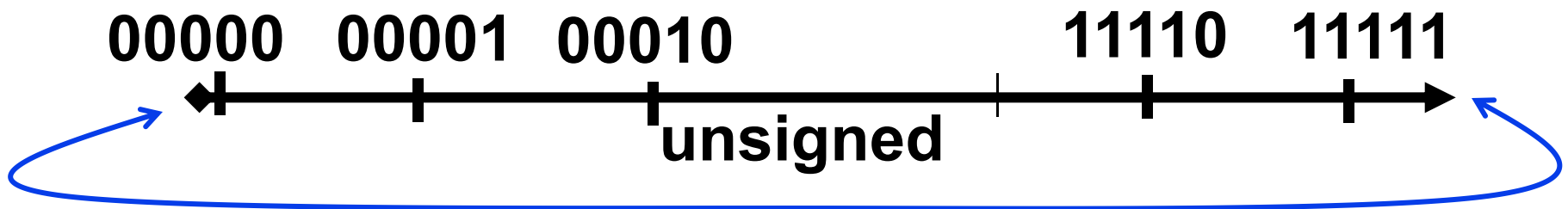
$$\begin{array}{cccc}
 & 1 & 1 & & \\
 & 1 & 0 & 1 & 0 \\
 + & 0 & 1 & 1 & 1 \\
 \hline
 1 & 0 & 0 & 0 & 1
 \end{array}$$

- **Example: $10 + 7 = 17$**

- ...so simple to add in binary that we can build circuits to do it!
- subtraction just as you would in decimal
- Comparison: How do you tell if $X > Y$?

What if too big?

- Binary bit patterns above are simply representatives of numbers. Strictly speaking they are called “numerals”.
- Numbers really have an ∞ number of digits
 - with almost all being same (00...0 or 11...1) except for a few of the rightmost digits
 - Just don't normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.



How to Represent Negative Numbers?

(C's unsigned int, C99's uintN_t)

- So far, unsigned numbers

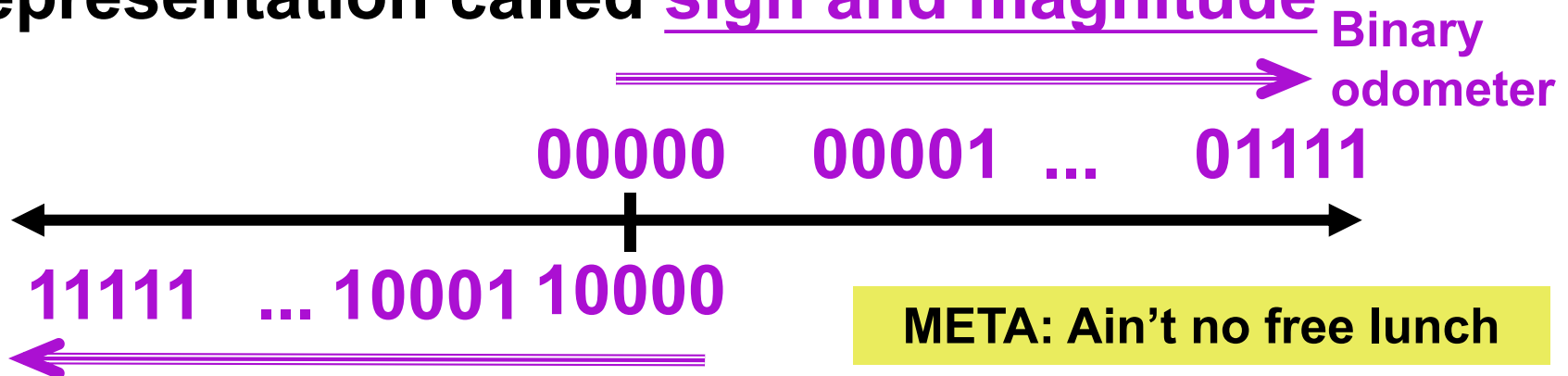


- Obvious solution: define leftmost bit to be sign!

• 0 → + 1 → -

• Rest of bits can be numerical value of number

- Representation called sign and magnitude



META: Ain't no free lunch

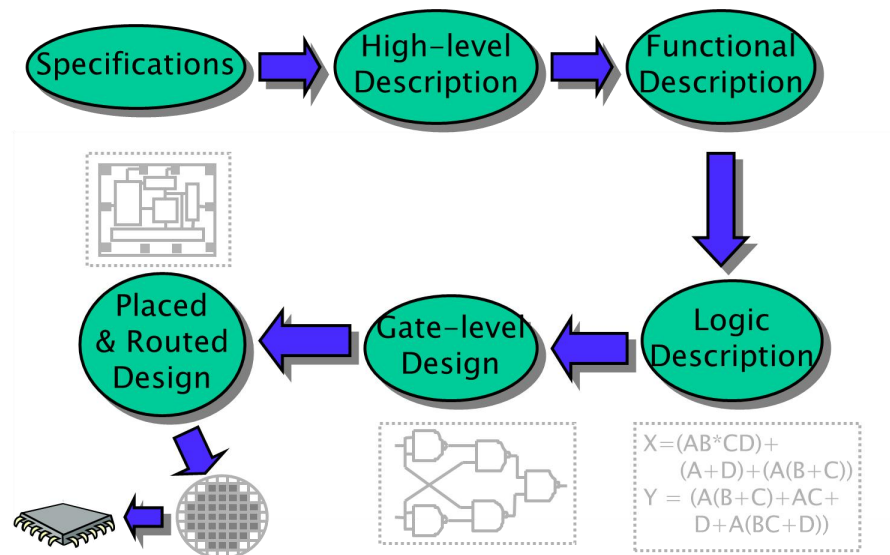
Shortcomings of sign and magnitude?

- **Arithmetic circuit complicated**
 - Special steps depending whether signs are the same or not
- **Also, two zeros**
 - $0x00000000 = +0_{\text{ten}}$
 - $0x80000000 = -0_{\text{ten}}$
 - What would two 0s mean for programming?
- **Also, incrementing “binary odometer”, sometimes increases values, and sometimes decreases!**
- **Therefore sign and magnitude abandoned**

Great EDA course I supervise

- **Introduction to VLSI Design Automation**

- The first EDA course in Beihang University
- Learn physical design or design automation of ICs
- Prereqs (data structures, programming language, algorithms, VLSI design)
- <http://www.cadetlab.cn/courses>

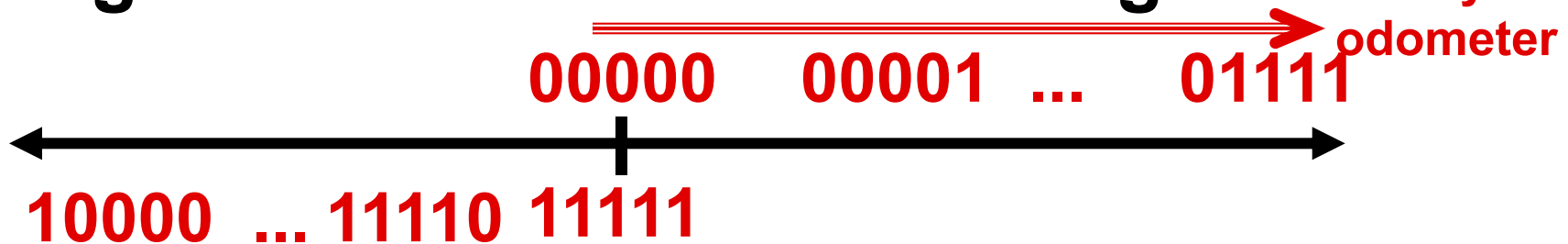


Another try: complement the bits

- Example: $7_{10} = 00111_2$ $-7_{10} = 11000_2$

- Called One's Complement

- Note: positive numbers have leading 0s, negative numbers have leading 1s.



- What is -00000 ? Answer: 11111

- How many positive numbers in N bits?

- How many negative numbers?

Shortcomings of One's complement?

- Arithmetic still a somewhat complicated.
- Still two zeros
 - $0x00000000 = +0_{\text{ten}}$
 - $0xFFFFFFF = -0_{\text{ten}}$
- Although used for a while on some computer products, one's complement was eventually abandoned because another solution was better.

Standard Negative # Representation

- Problem is the negative mappings “overlap” with the positive ones (the two 0s). Want to shift the negative mappings left by one.
 - Solution! For negative numbers, complement, then add 1 to the result
- As with sign and magnitude, & one’s compl. leading 0s is positive, leading 1s is negative
 - 000000...xxx is ≥ 0 , 111111...xxx is < 0
 - except 1...1111 is -1, not -0 (as in sign & mag.)
- This representation is **Two’s Complement**
 - This makes the hardware simple!

(C’s `int`, aka a “signed integer”)
(Also C’s `short`, `long long`, ..., C99’s `intN_t`)

Two's Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:

$$d_{31} \times \textcolor{red}{-(2^{31})} + d_{30} \times 2^{30} + \dots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$$

- Example: 1101_{two} in a nibble?

$$= \textcolor{red}{1} \times \textcolor{red}{-(2^3)} + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= \textcolor{red}{-2^3} + 2^2 + 0 + 2^0$$

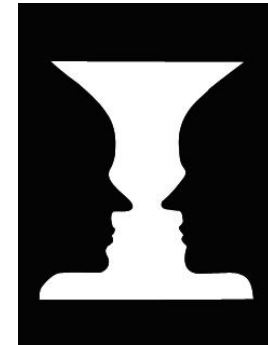
$$= \textcolor{red}{-8} + 4 + 0 + 1$$

$$= \textcolor{red}{-8} + 5$$

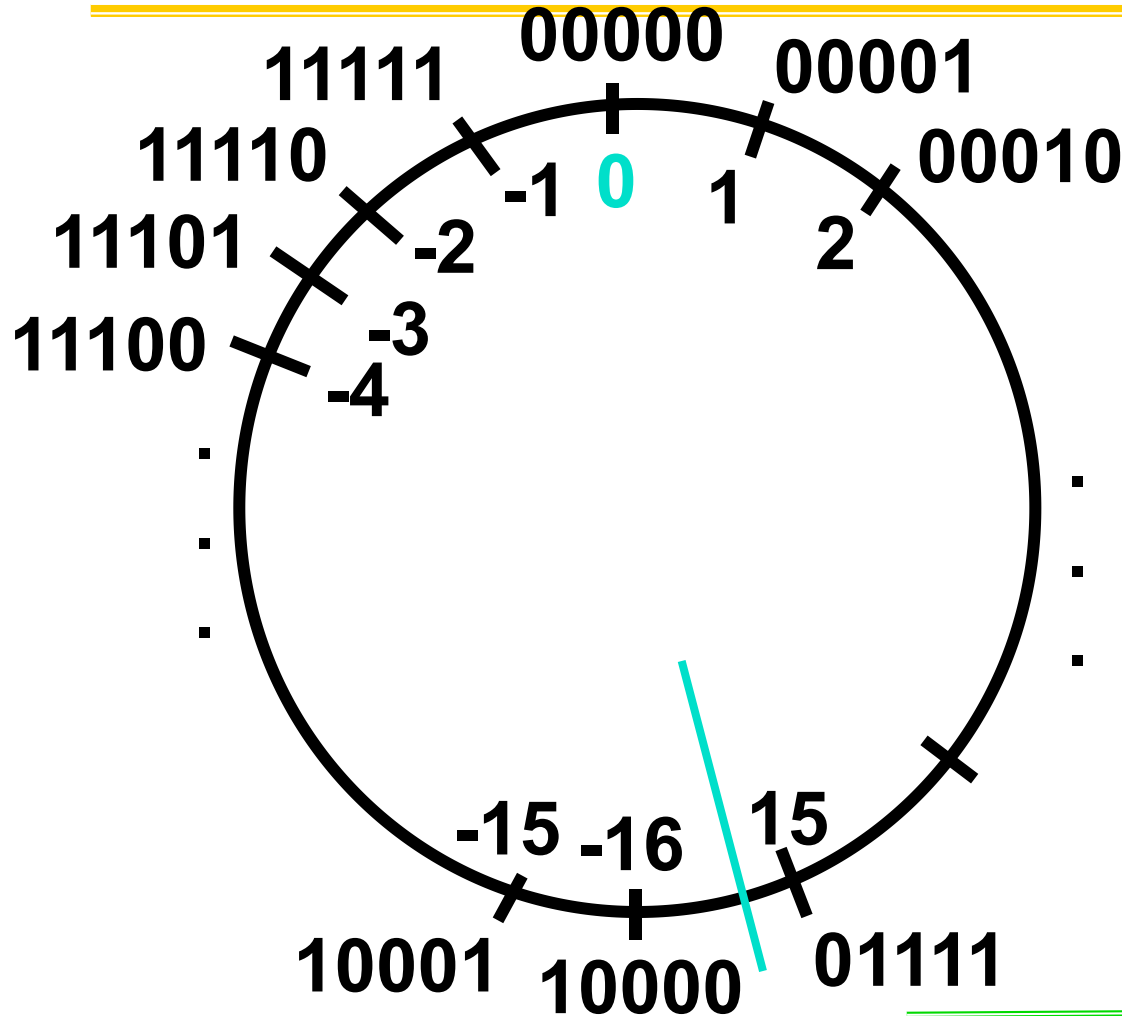
$$= \textcolor{red}{-3}_{\text{ten}}$$

Example: -3 to +3 to -3
(again, in a nibble):

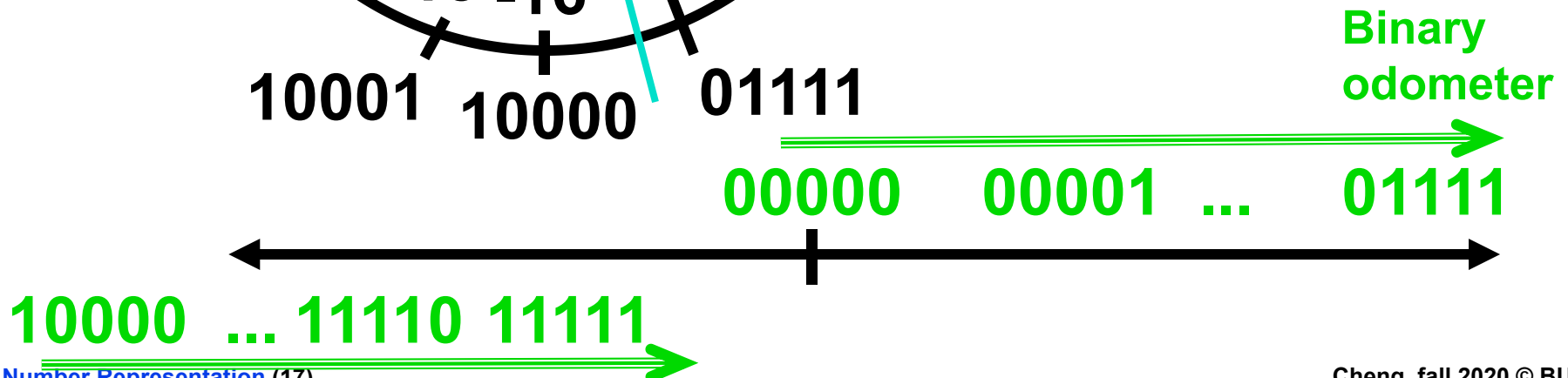
| | | |
|-------|------|----------------|
| x : | 1101 | _{two} |
| x' : | 0010 | _{two} |
| +1 : | 0011 | _{two} |
| ()' : | 1100 | _{two} |
| +1 : | 1101 | _{two} |



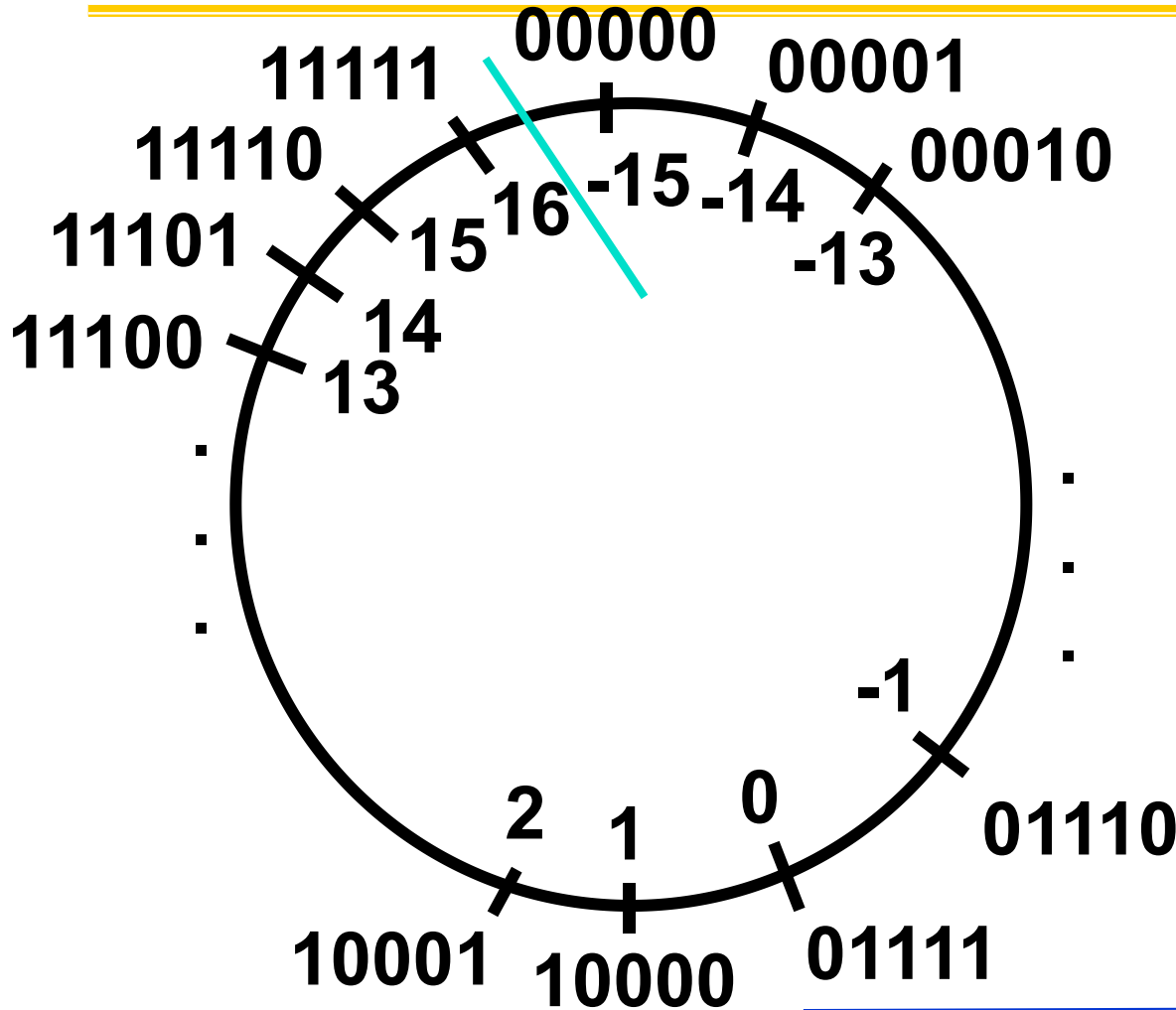
2's Complement Number "line": $N = 5$



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- **one zero**
- how many positives?



Bias Encoding: $N = 5$ (bias = -15)

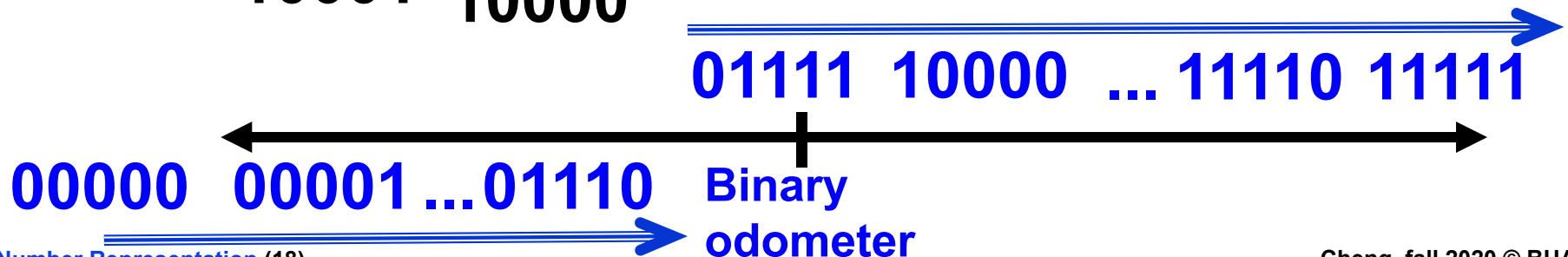


- # = unsigned + bias

- Bias for N bits chosen as $-(2^{N-1}-1)$

- one zero

- how many positives?



How best to represent -12.75?



- a) 2s Complement (but shift binary pt)
- b) Bias (but shift binary pt)
- c) Combination of 2 encodings
- d) Combination of 3 encodings
- e) We can't

Shifting binary point means “divide number by some power of 2. E.g.,
 $11_{10} = 1011.0_2 \rightarrow 10.110_2 = (11/4)_{10} = 2.75_{10}$

And in summary...

META: We often make design decisions to make HW simple

- We represent “things” in computers as particular bit patterns: N bits $\Rightarrow 2^N$ things
- These 5 integer encodings have different benefits; 1s complement and sign/mag have most problems.
- **unsigned** (C99's `uintN_t`) :

00000 00001 ... 01111 10000 ... 11111



- **2's complement** (C99's `intN_t`) universal, learn!

00000 00001 ... 01111
10000 ... 11110 11111



- Overflow: numbers ∞ ; computers finite, errors!

META: Ain't no free lunch

REFERENCE: Which base do we use?

- **Decimal:** great for humans, especially when doing arithmetic
- **Hex:** if human looking at long strings of binary numbers, its much easier to convert to hex and look 4 bits/symbol
 - Terrible for arithmetic on paper
- **Binary:** what computers use; you will learn how computers do +, -, *, /
 - To a computer, numbers always binary
 - Regardless of how number is written:
 - $32_{\text{ten}} == 32_{10} == 0x20 == 100000_2 == 0b100000$
 - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing

Two's Complement for N=32

| | | |
|------------------------------|------------------|-------------------------------|
| 0000 ... 0000 0000 0000 0000 | _{two} = | 0 _{ten} |
| 0000 ... 0000 0000 0000 0001 | _{two} = | 1 _{ten} |
| 0000 ... 0000 0000 0000 0010 | _{two} = | 2 _{ten} |
| 0111 ... 1111 1111 1111 1101 | _{two} = | 2,147,483,645 _{ten} |
| 0111 ... 1111 1111 1111 1110 | _{two} = | 2,147,483,646 _{ten} |
| 0111 ... 1111 1111 1111 1111 | _{two} = | 2,147,483,647 _{ten} |
| 1000 ... 0000 0000 0000 0000 | _{two} = | -2,147,483,648 _{ten} |
| 1000 ... 0000 0000 0000 0001 | _{two} = | -2,147,483,647 _{ten} |
| 1000 ... 0000 0000 0000 0010 | _{two} = | -2,147,483,646 _{ten} |
| 1111 ... 1111 1111 1111 1101 | _{two} = | -3 _{ten} |
| 1111 ... 1111 1111 1111 1110 | _{two} = | -2 _{ten} |
| 1111 ... 1111 1111 1111 1111 | _{two} = | -1 _{ten} |

- One zero; 1st bit called sign bit
- 1 “extra” negative: no positive 2,147,483,648_{ten}

Two's comp. shortcut: Sign extension

- Convert 2's complement number rep. using n bits to more than n bits
- Simply **replicate** the most significant bit (sign bit) of smaller to fill new bits
 - 2's comp. positive number has infinite 0s
 - 2's comp. negative number has infinite 1s
 - Binary representation hides leading bits; sign extension restores some of them
 - 16-bit -4_{ten} to 32-bit:

1111 1111 1111 1100_{two}

1111 1111 1111 1111 1111 1111 1111 1100_{two}

Chapter 2 Week2: MIPS

2.1 MIPS Arithmetic

Computer Architecture (计算机体系结构)

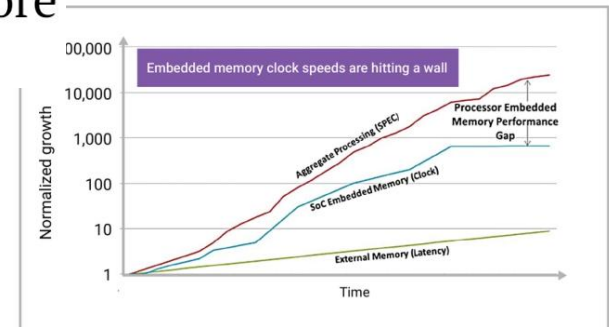
Lecture #3 – Introduction to MIPS Assembly language : Arithmetic



Lecturer Yuanqing Cheng (成元庆)

www.cadetlab.cn

Meeting Increasing Performance
Requirements in Embedded
Applications with Scalable Multicore
Processors



Review

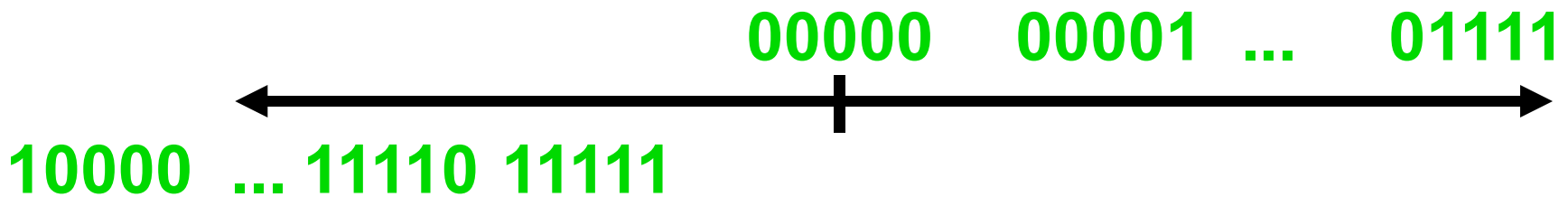
META: We often make design decisions to make HW simple

- We represent “things” in computers as particular bit patterns: N bits $\Rightarrow 2^N$ things
- These 5 integer encodings have different benefits; 1s complement and sign/mag have most problems.
- **unsigned** (C99's `uint N _t`) :

00000 00001 ... 01111 10000 ... 11111



- **2's complement** (C99's `int N _t`) universal, learn!



- Overflow: numbers ∞ ; computers finite, errors!

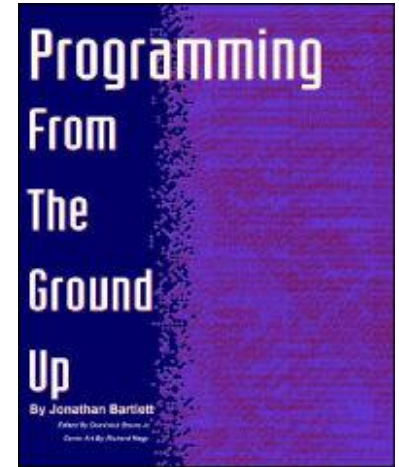
META: Ain't no free lunch

Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture (ISA)*.
 - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...

Book: *Programming From the Ground Up*

*“A new book was just released which is based on a new concept - teaching computer science through assembly language (Linux x86 assembly language, to be exact). This book teaches how the machine itself operates, rather than just the language. I've found that the key difference between mediocre and excellent programmers is whether or not they know assembly language. **Those that do tend to understand computers themselves at a much deeper level.***



*Although [almost!] unheard of today, this concept isn't really all that new -- there used to not be much choice in years past. Apple computers came with only BASIC and assembly language, and there were books available on assembly language for kids. This is why the old-timers are often viewed as 'wizards': they **had** to know assembly language programming.”*

-- slashdot.org comment, 2004-02-05

Instruction Set Architectures

- **Early trend was to add more and more instructions to new CPUs to do elaborate operations**
 - **VAX architecture had an instruction to multiply polynomials!**
- **RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing**
 - **Keep the instruction set small and simple, makes it easier to build fast hardware.**
 - **Let software do complicated operations by composing simpler ones.**

MIPS Architecture

- **MIPS** – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses CS 152, 162, 164)
- **Why MIPS instead of Intel 80x86?**
 - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
 - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs



Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

Assembly Variables: Registers (1/4)

- **Unlike HLL like C or Java, assembly cannot use variables**
 - **Why not? Keep Hardware Simple**
- **Assembly Operands are registers**
 - **limited number of special locations built directly into the hardware**
 - **operations can only be performed on these!**
- **Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)**

Assembly Variables: Registers (2/4)

- **Drawback:** Since registers are in hardware, there are a predetermined number of them
 - **Solution:** MIPS code must be very carefully put together to efficiently use registers
- **32 registers in MIPS**
 - **Why 32? Smaller is faster**
- **Each MIPS register is 32 bits wide**
 - **Groups of 32 bits called a word in MIPS**

Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:

`$0, $1, $2, ... $30, $31`

Assembly Variables: Registers (4/4)

- By convention, each register also has a name to make it easier to code

- For now:

$\$16 - \$23 \rightarrow \$s0 - \$s7$

(correspond to C variables)

$\$8 - \$15 \rightarrow \$t0 - \$t7$

(correspond to temporary variables)

Later will explain other 16 register names

- In general, use names to make your code more readable

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type
 - Example:
`int fahr, celsius;`
`char a, b, c, d, e;`
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for MIPS comments
 - anything from hash mark to end of line is a comment and will be ignored
 - This is just like the C99 //
- Note: Different from C.
 - C comments have format
/* comment */
so they can span many lines

Assembly Instructions

- In assembly language, each statement (called an **Instruction**), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- **Ok, enough already...gimme my MIPS!**

MIPS Addition and Subtraction (1/4)

- **Syntax of Instructions:**

1 2,3,4

where:

1) operation by name

2) operand getting result (“destination”)

3) 1st operand for operation (“source1”)

4) 2nd operand for operation (“source2”)

- **Syntax is rigid:**

- 1 operator, 3 operands

- **Why? Keep Hardware simple via regularity**

Addition and Subtraction of Integers (2/4)

- **Addition in Assembly**

- **Example:** `add $s0,$s1,$s2` (in MIPS)

- Equivalent to: $a = b + c$ (in C)

- where MIPS registers `$s0`, `$s1`, `$s2` are associated with C variables `a`, `b`, `c`

- **Subtraction in Assembly**

- **Example:** `sub $s3,$s4,$s5` (in MIPS)

- Equivalent to: $d = e - f$ (in C)

- where MIPS registers `$s3`, `$s4`, `$s5` are associated with C variables `d`, `e`, `f`

Addition and Subtraction of Integers (3/4)

- How do the following C statement?

`a = b + c + d - e;`

- Break into multiple instructions

`add $t0, $s1, $s2 # temp = b + c`

`add $t0, $t0, $s3 # temp = temp + d`

`sub $s0, $t0, $s4 # a = temp - e`

- Notice: A single line of C may break up into several lines of MIPS.
- Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers (4/4)

- How do we do this?

$$f = (g + h) - (i + j);$$

- Use intermediate temporary register

| | |
|-----------------------------------|----------------------------------|
| <code>add \$t0, \$s1, \$s2</code> | <code># temp = g + h</code> |
| <code>add \$t1, \$s3, \$s4</code> | <code># temp = i + j</code> |
| <code>sub \$s0, \$t0, \$t1</code> | <code># f = (g+h) - (i+j)</code> |

Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero (\$0 or **\$zero**) to always have the value 0; eg

`add $s0, $s1, $zero` (in MIPS)

`f = g` (in C)

where MIPS registers `$s0, $s1` are associated with C variables `f, g`

- defined in hardware, so an instruction

`add $zero, $zero, $s0`

will not do anything!

Immediates

- **Immediates are numerical constants.**
- **They appear often in code, so there are special instructions for them.**

- **Add Immediate:**

`addi $s0,$s1,10` (in MIPS)

`f = g + 10` (in C)

where MIPS registers `$s0`, `$s1` are associated with C variables `f`, `g`

- **Syntax similar to add instruction, except that last argument is a number instead of a register.**

Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
 - if an operation can be decomposed into a simpler operation, don't include it
 - `addi ..., -X = subi ..., X` \Rightarrow so no `subi`
- `addi $s0,$s1,-10` (in MIPS)
 $f = g - 10$ (in C)
where MIPS registers `$s0`, `$s1` are associated with C variables `f`, `g`

Peer Instruction

- 1) Since there are only 8 local (\$s) and 8 temp (\$t) variables, we can't write MIPS for C exprs that contain > 16 vars.
- 2) If p (stored in \$s0) were a pointer to an array of ints, then p++; would be
addi \$s0 \$s0 1

| | |
|----|-------|
| | 12 |
| a) | FF |
| b) | FT |
| c) | TF |
| d) | TT |
| e) | dunno |

“And in Conclusion...”

- **In MIPS Assembly Language:**
 - Registers replace C variables
 - One Instruction (simple operation) per line
 - Simpler is Better
 - Smaller is Faster
- **New Instructions:**
add, addi, sub
- **New Registers:**
C Variables: \$s0 - \$s7
Temporary Variables: \$t0 - \$t9
Zero: \$zero

2.2 Data Transfer & Decisions



Lecturer
Yuanqing
Cheng

Computer Architecture (计算机体系结构)

Lecture 4 – Introduction to MIPS Data Transfer & Decisions I

2020-09-11



Review

- In MIPS Assembly Language:
 - Registers replace variables
 - One Instruction (simple operation) per line
 - Simpler is Better, Smaller is Faster
- New Instructions:
`add, addi, sub`
- New Registers:
C Variables: `$s0 - $s7`
Temporary Variables: `$t0 - $t7`
Zero: `$zero`

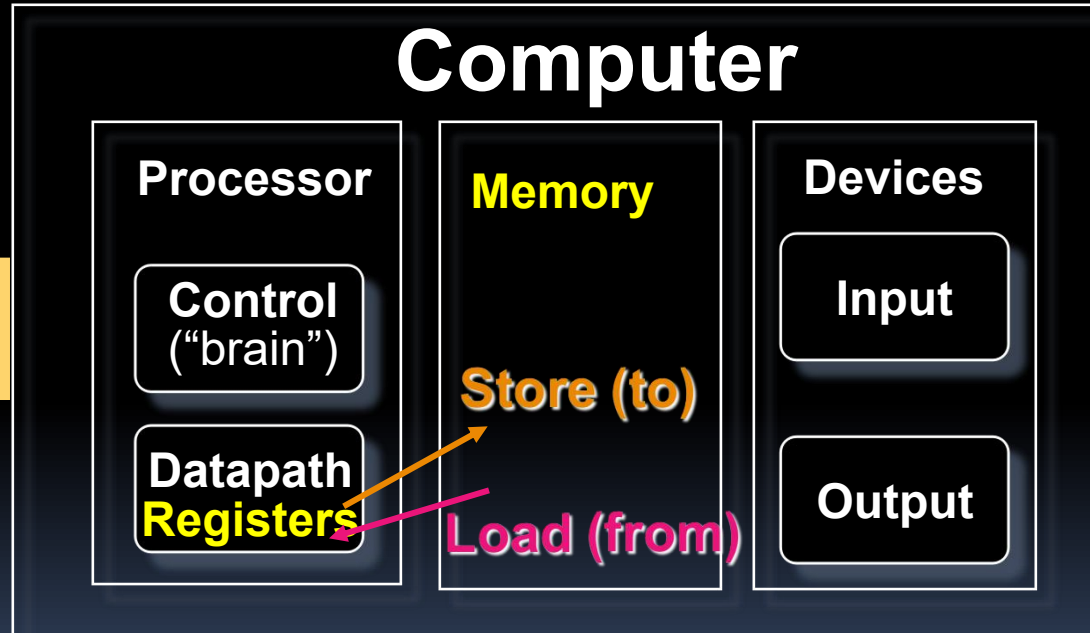
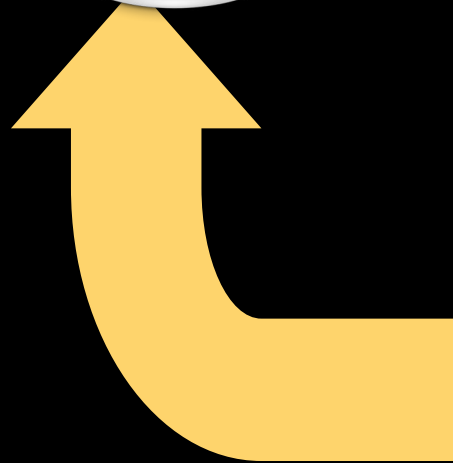
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: **memory** contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- **Data transfer instructions** transfer data between registers and memory:
 - **Memory to register**
 - **Register to memory**

Anatomy: 5 components of any Computer



Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.



These are “data transfer” instructions...

Data Transfer: Memory to Reg (1/4)

- To transfer a word of data, we need to specify two things:
 - **Register:** specify this by # (\$0 - \$31) or symbolic name (\$s0, ..., \$t0, ...)
 - **Memory address:** more difficult
 - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
 - Other times, we want to be able to **offset** from this pointer.
- Remember: “**Load FROM memory**”

Data Transfer: Memory to Reg (2/4)

- To specify a memory address to copy from, specify two things:
 - A register containing a pointer to memory
 - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.
- Example: 8 (\$t0)
 - specifies the memory address pointed to by the value in \$t0, plus 8 bytes

Data Transfer: Memory to Reg (3/4)

- Load Instruction Syntax:

1 2, 3 (4)

- where

- 1) operation name

- 2) register that will receive value

- 3) numerical offset in bytes

- 4) register containing pointer to memory

- MIPS Instruction Name:

- **lw** (meaning Load Word, so 32 bits or one word are loaded at a time)

Data Transfer: Memory to Reg (4/4)



Example: `lw $t0, 12($s0)`

This instruction will take the pointer in `$s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`

■ Notes:

- `$s0` is called the base register
- `12` is called the offset
- offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure (note offset must be a **constant known at assembly time**)

Data Transfer: Reg to Memory

- Also want to store from register into memory
 - Store instruction syntax is identical to Load's
- MIPS Instruction Name:
sw (meaning Store Word, so 32 bits or one word is stored at a time)

Data flow



- Example: **sw \$t0, 12(\$s0)**


This instruction will take the pointer in \$s0, add 12 bytes to it, and then store the value from register \$t0 into that memory address

- Remember: “Store INTO memory”

Pointers v. Values

- **Key Concept:** A register can hold any 32-bit value. That value can be a (signed) `int`, an unsigned `int`, a pointer (memory addr), and so on
 - E.g., If you write: `add $t2,$t1,$t0` then `$t0` and `$t1` better contain values that can be added
 - E.g., If you write: `lw $t2,0($t0)` then `$t0` better contain a pointer
- Don't mix these up!

Addressing: Byte vs. Word

- Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
 - `Memory[0]`, `Memory[1]`, `Memory[2]`, ...

Called the “address” of a word
- Computers needed to access 8-bit bytes as well as words (4 bytes/word)
- Today machines address memory as bytes, (i.e., “**Byte Addressed**”) hence 32-bit (4 byte) word addresses differ by 4
 - `Memory[0]`, `Memory[4]`, `Memory[8]`

Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
- $4 \times 5 = 20$ to select `A[5]`: byte v. word
- Compile by hand using registers:

`g = h + A[5];`

- `g: $s1, h: $s2, $s3: base address of A`
- 1st transfer from memory to register:

`lw $t0, 20($s3) # $t0 gets A[5]`

- Add 20 to `$s3` to select `A[5]`, put into `$t0`

- Next add it to `h` and place in `g`

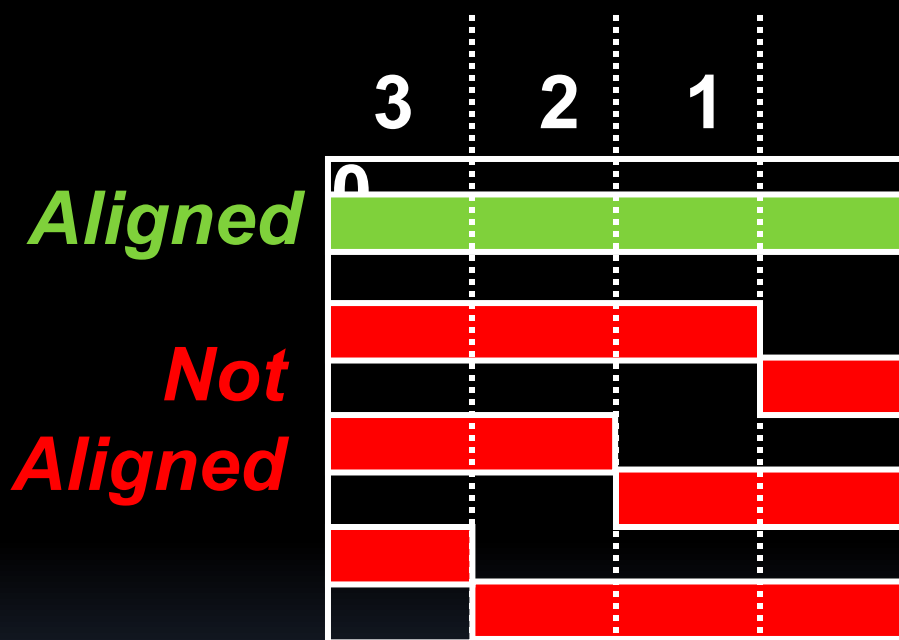
`add $s1, $s2, $t0 # $s1 = h + A[5]`

Notes about Memory

- **Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.**
 - **Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.**
 - **Also, remember that for both `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)**

More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes



Last hex digit
of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

- Called Alignment: objects fall on address that is multiple of their size

Role of Registers vs. Memory

- What if more variables than registers?
 - Compiler tries to keep most frequently used variable in registers
 - Less common variables in memory: spilling
- Why not keep all variables in memory?
 - Smaller is faster:
registers are faster than memory
 - Registers more versatile:
 - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
 - MIPS data transfer only read or write 1 operand per instruction, and no operation

So Far...

- All instructions so far only manipulate data...we've built a **calculator** of sorts.
- In order to build a **computer**, we need ability to make decisions...
- C (and MIPS) provide labels to support “**goto**” jumps to places in code.
 - **C: Horrible style; MIPS: Necessary!**
- Heads up: pull out some papers and pens, you'll do an in-class exercise!

C Decisions: `if` Statements

- 2 kinds of `if` statements in C

`if (condition) clause`

`if (condition) clause1 else clause2`

- Rearrange 2nd `if` into following:

`if (condition) goto L1;`

`clause2;`

`goto L2;`

`L1: clause1;`

`L2:`

- Not as elegant as `if-else`, but same meaning

MIPS Decision Instructions

- Decision instruction in MIPS:

`beq register1, register2, L1`

`beq` is “Branch if (registers are) equal”

Same meaning as (using C):

`if (register1==register2) goto L1`

- Complementary MIPS decision instruction

`bne register1, register2, L1`

`bne` is “Branch if (registers are) not equal”

Same meaning as (using C):

`if (register1!=register2) goto L1`

- Called conditional branches

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an unconditional branch:

`j label`

- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition
- Same meaning as (using C): `goto label`
- Technically, it's the same effect as:
`beq $0,$0,label`
since it always satisfies the condition.

Compiling C `if` into MIPS (1/2)

- Compile by hand

```
if (i == j) f=g+h;  
else f=g-h;
```

- Use this mapping:

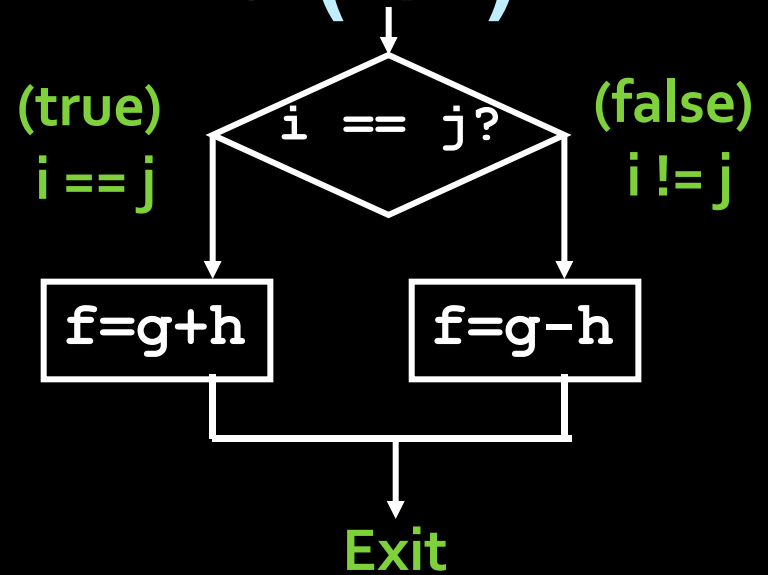
f: \$s0

g: \$s1

h: \$s2

i: \$s3

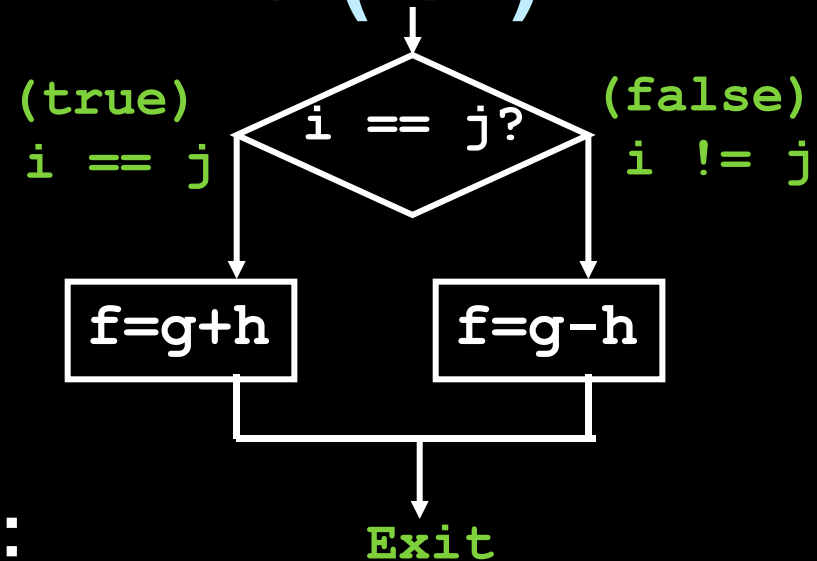
j: \$s4



Compiling C `if` into MIPS (2/2)

- Compile by hand

```
if (i == j) f=g+h;  
else f=g-h;
```



- Final compiled MIPS code:

```
        beq $s3,$s4,True    # branch i==j  
        sub $s0,$s1,$s2    # f=g-h (false)  
        j    Fin           # goto Fin  
True:   add $s0,$s1,$s2    # f=g+h (true)  
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

Peer

We want to translate $*x = *y$ into MIPS
(x, y ptrs stored in: $\$s0$ $\$s1$)

```
1: add $s0, $s1, zero
2: add $s1, $s0, zero
3: lw  $s0, 0($s1)
4: lw  $s1, 0($s0)
5: lw  $t0, 0($s1)
6: sw  $t0, 0($s0)
7: lw  $s0, 0($t0)
8: sw  $s1, 0($t0)
```

- a) 1 or 2
- b) 3 or 4
- c) 5→6
- d) 6→5
- e) 7→8

“And in Conclusion...”

- Memory is **byte**-addressable, but **lw** and **sw** access one **word** at a time.
- A pointer (used by **lw** and **sw**) is just a memory address, we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using **conditional statements** within **if**, **while**, **do while**, **for**.
- MIPS Decision making instructions are the **conditional branches**: **beq** and **bne**.
- New Instructions:
lw, sw, beq, bne, j