

# Machine Learning's Slides

**Author:** Pannenets.F

**Date:** October 9, 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](http://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



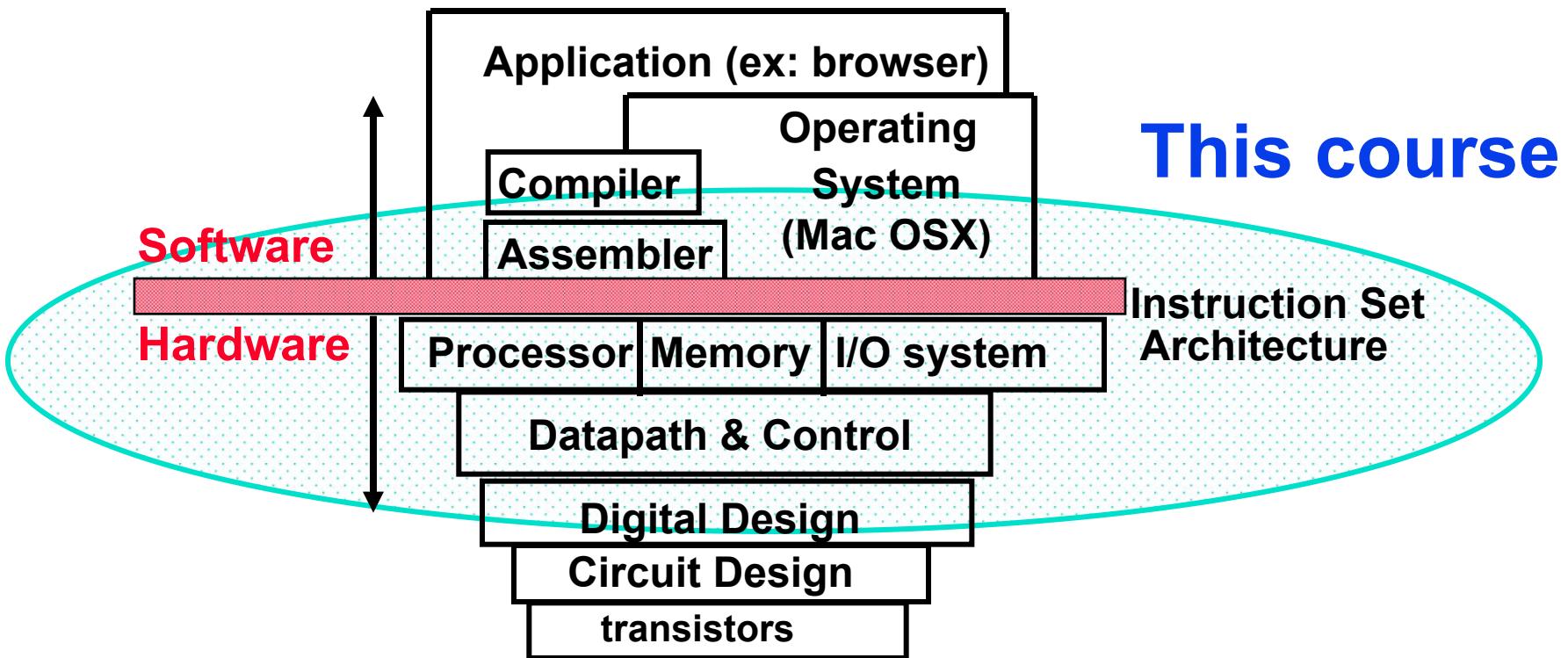
# Are Computers Smart?

---

- 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}

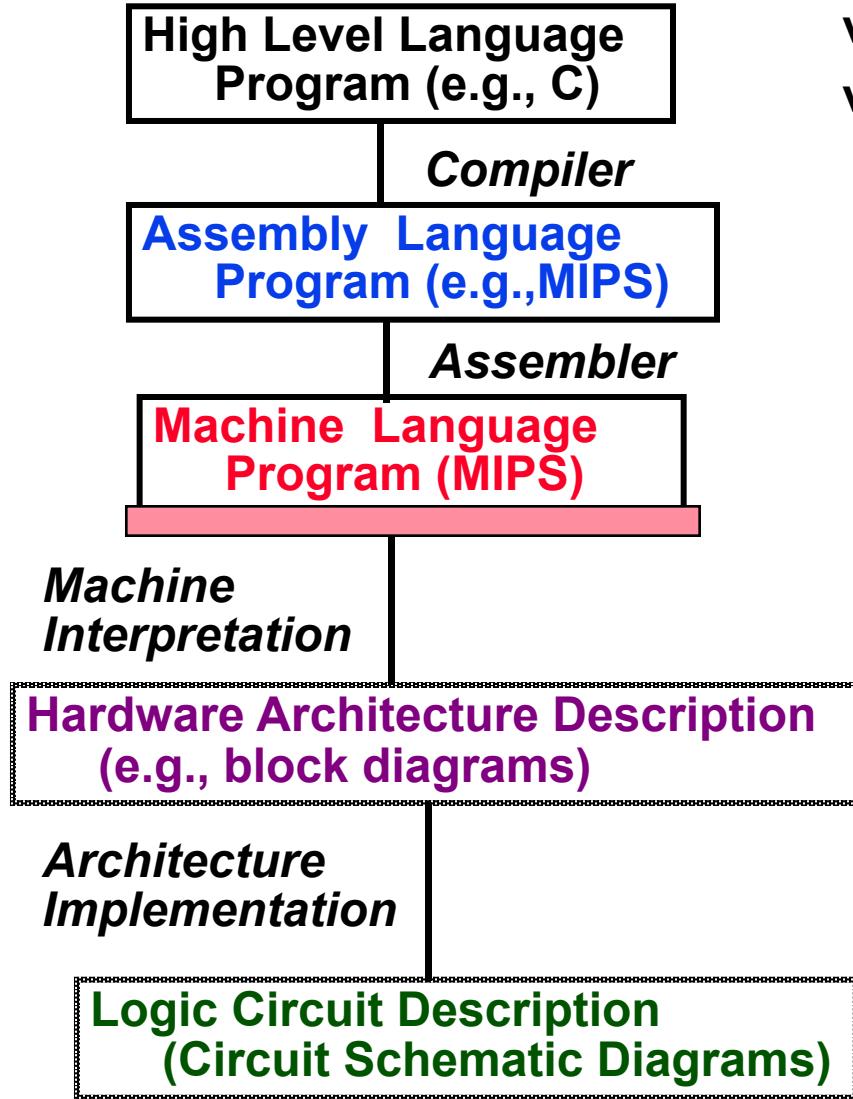


# What are “Computer Architecture”?



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

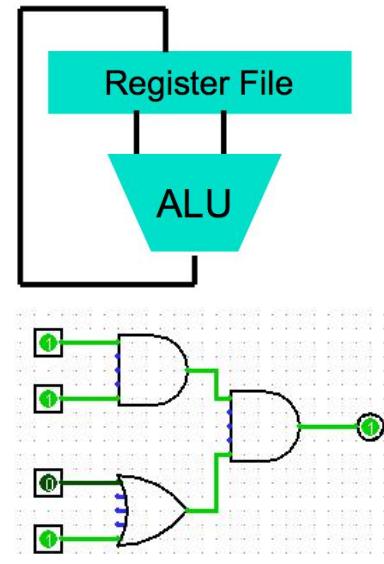
# Levels of Representation



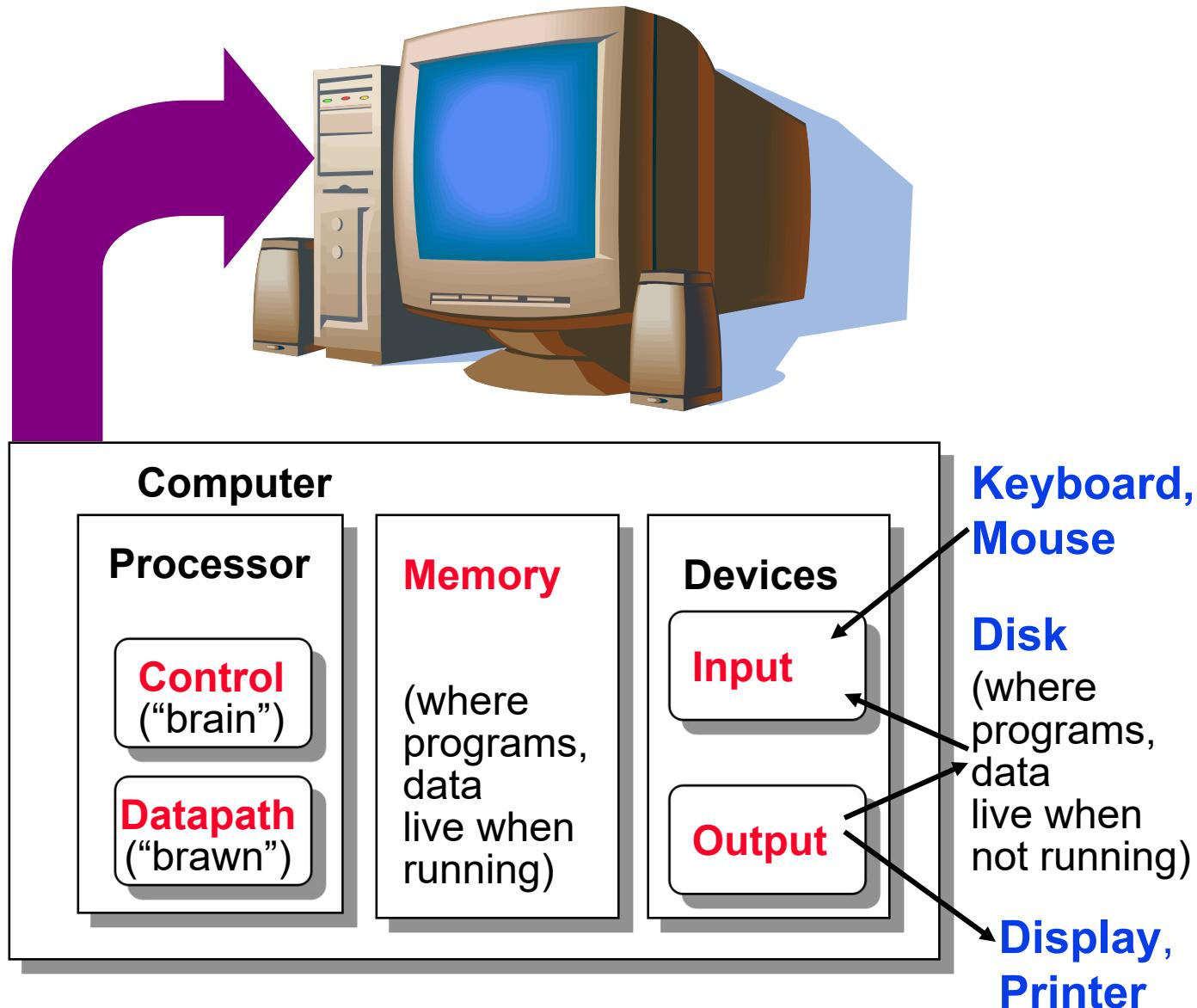
`temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;`

`Iw $t0, 0($2)  
Iw $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

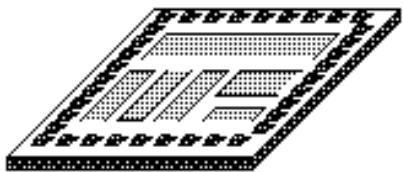
---

*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.**

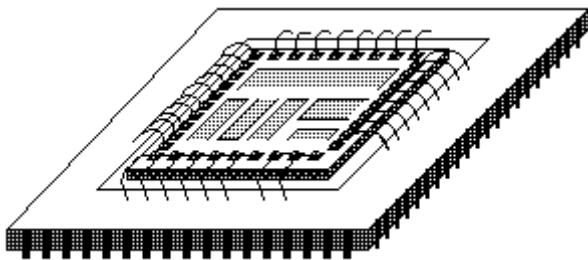
# Integrated Circuits (2020 state-of-the-art)

## Bare Die



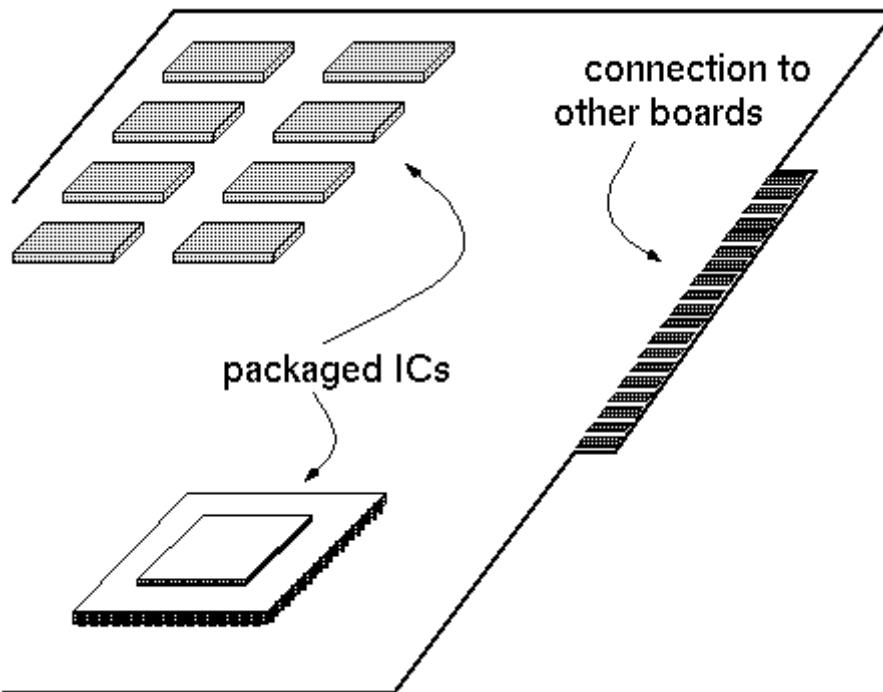
- Primarily Crystalline Silicon
- 1mm - 25mm on a side
- feature size  $\sim 14/7$  nm
- Billions of transistors
- (25 - 100M “logic gates”)
- 3 - 12 conductive layers
- “CMOS” (complementary metal oxide semiconductor) - most common.

## Chip in Package



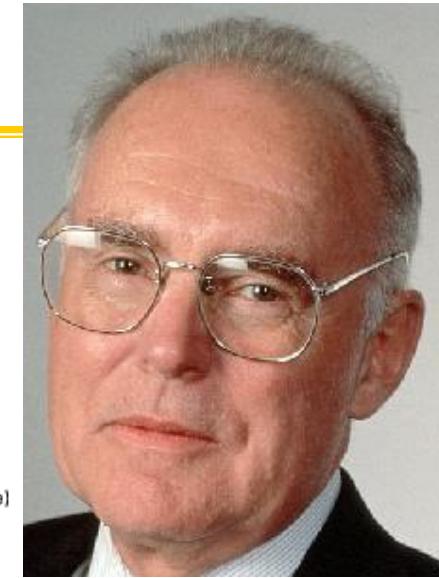
- Package provides:
  - spreading of chip-level signal paths to board-level
  - heat dissipation.
- Ceramic or plastic with gold wires.

# Printed Circuit Boards

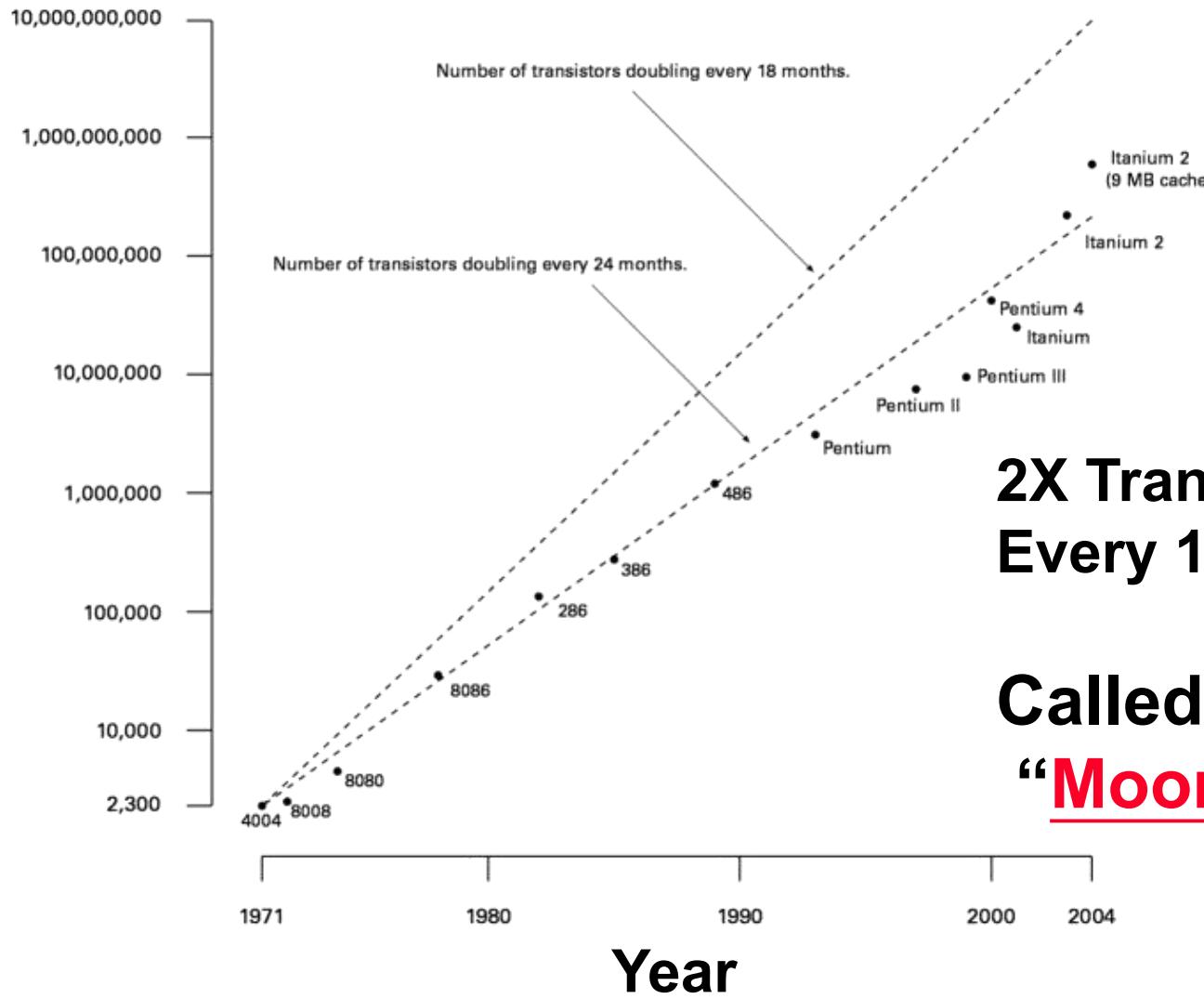


- **fiberglass or ceramic**
- **1-20 conductive layers**
- **1-20 in on a side**
- **IC packages are soldered down.**
- **Provides:**
  - Mechanical support
  - Distribution of power and heat.

# Technology Trends: Microprocessor Complexity



# of transistors on an IC

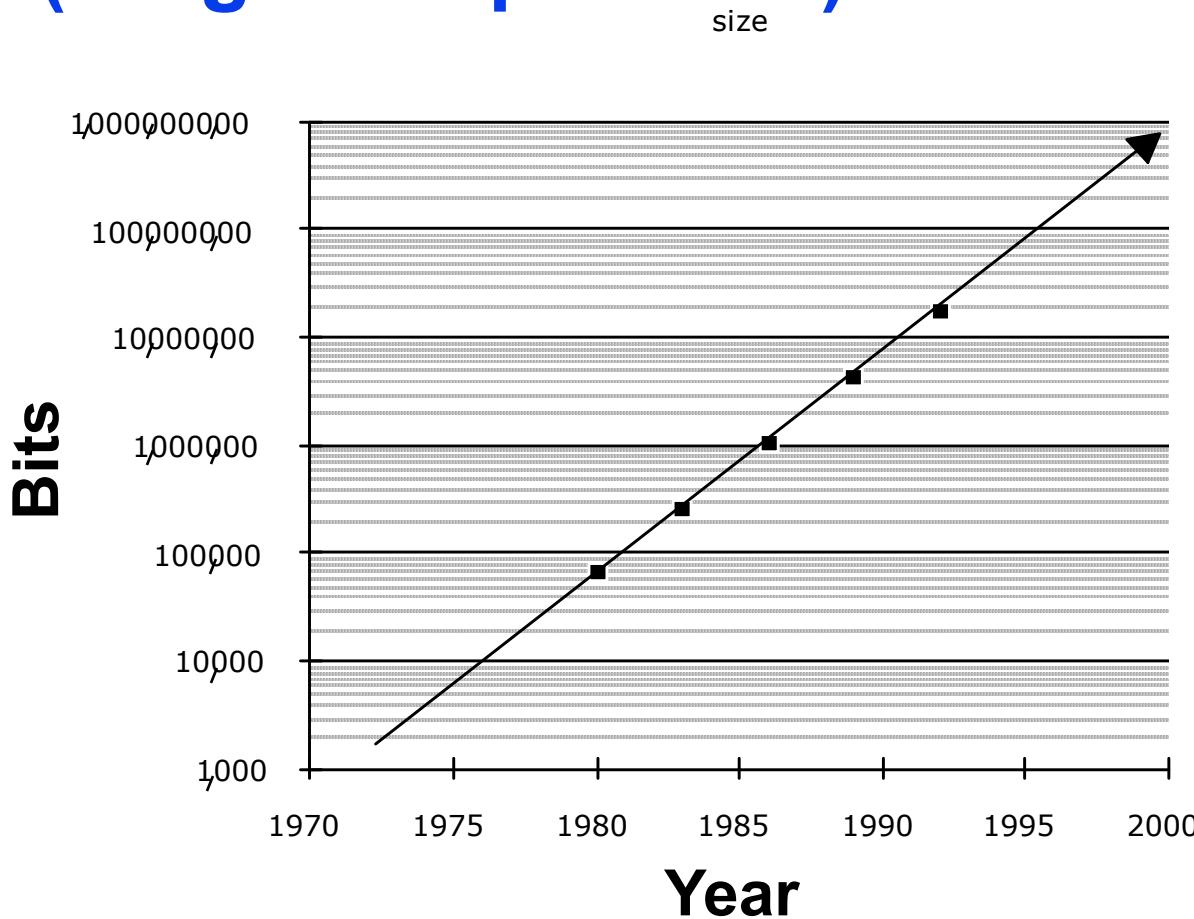


**Gordon Moore**  
Intel Cofounder!

**2X Transistors / Chip  
Every 1.5 years**

**Called  
“Moore’s Law”**

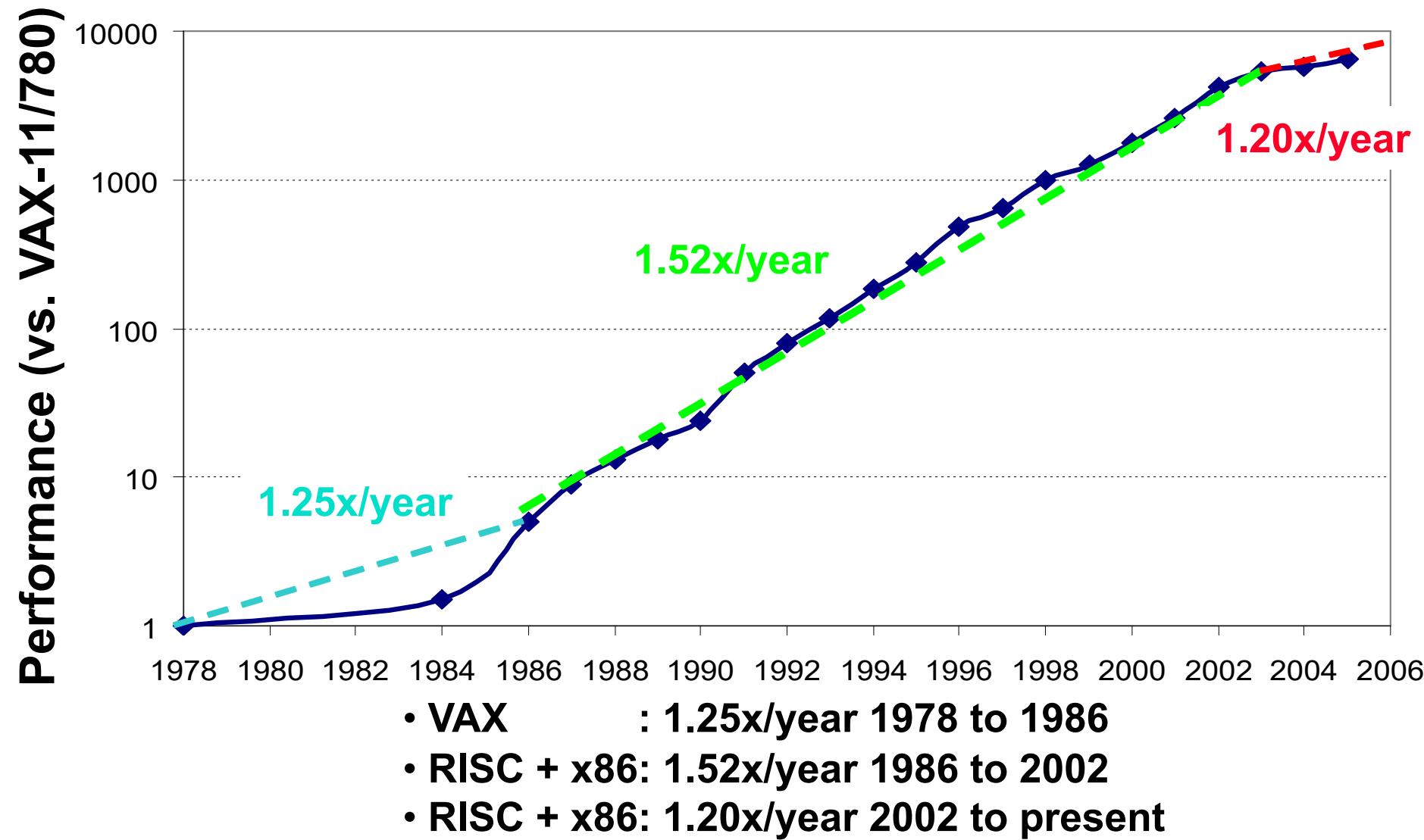
# Technology Trends: Memory Capacity (Single-Chip DRAM)



- 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
2006	2048 (2Gbit)
2010	8192 (8Gbit)

# Technology Trends: Uniprocessor Performance (SPECint)



# 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 **MegaHertz**  
(4.0 **GigaHertz**)
- Memory capacity:            65,536 **MebiBytes**  
(64.0 **GibiBytes**)
- Disk capacity:                2,000 **GigaBytes**  
(2.0 **TeraBytes**)
- New units! **Mega** ⇒ **Giga**, **Giga** ⇒ **Tera**  
  
(**Tera** ⇒ **Peta**, **Peta** ⇒ **Exa**, **Exa** ⇒ **Zetta**  
**Zetta** ⇒ **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**

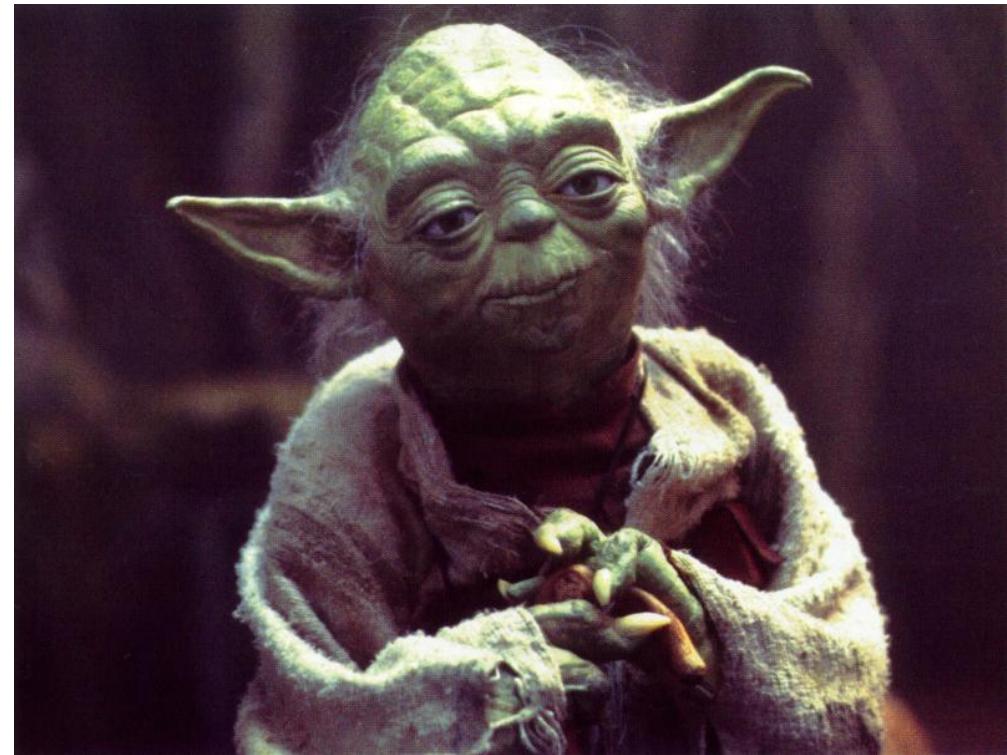
---

- **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...**

---

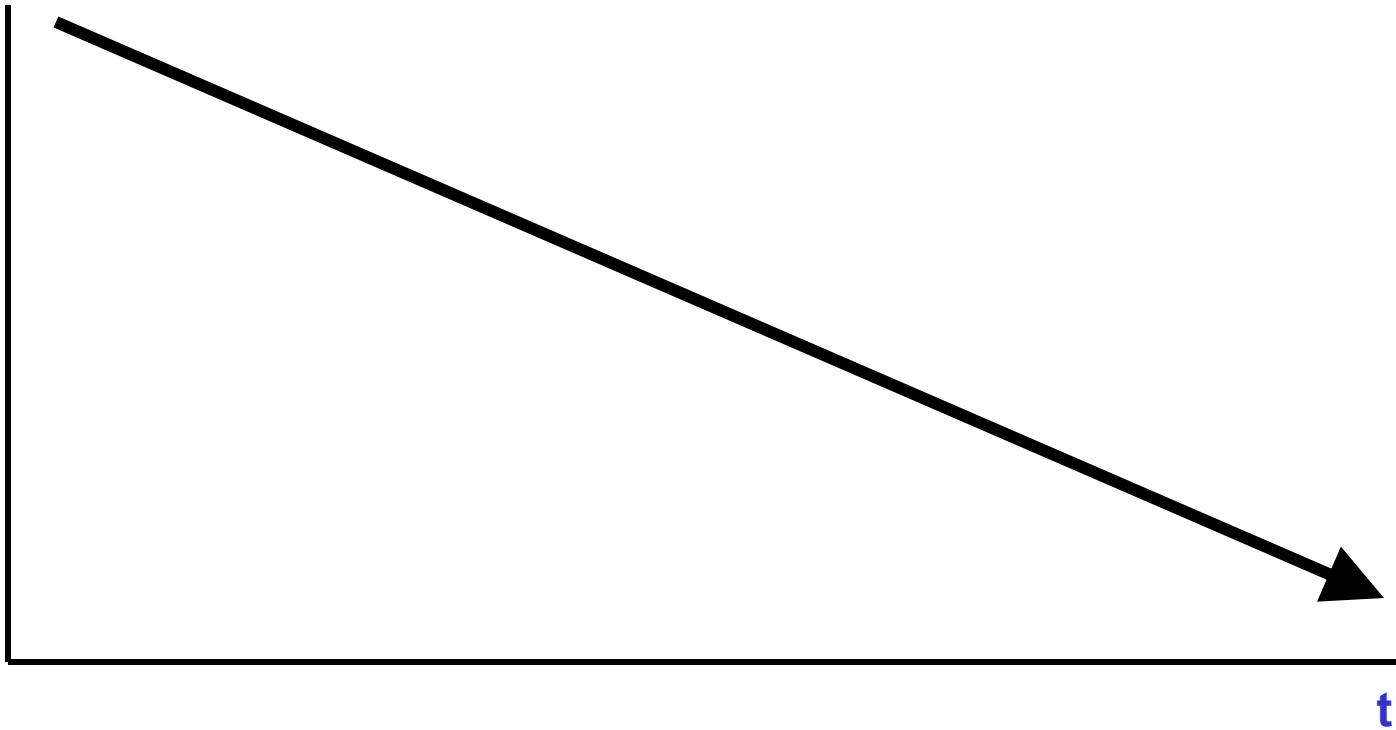
**“Always in  
motion is the  
future...”**



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

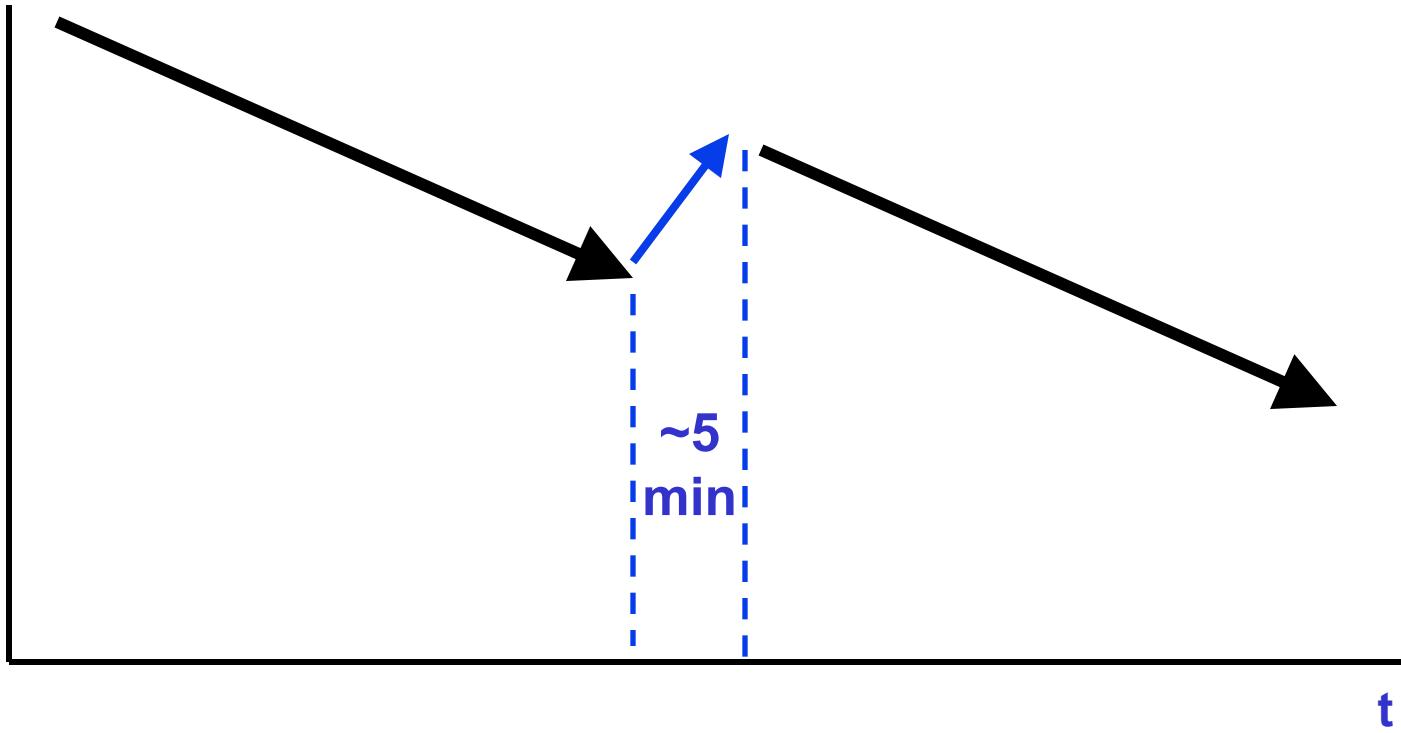
# What is this?

---



**Attention over time!**

# What is this?!

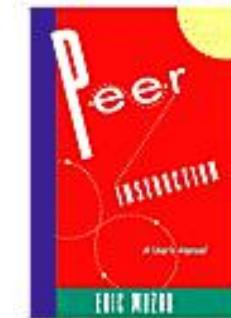


**Attention over time!**

# 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](http://www.cadetlab.cn/courses)
- We will DROP your lowest HW, Lab!
- Only one final exam

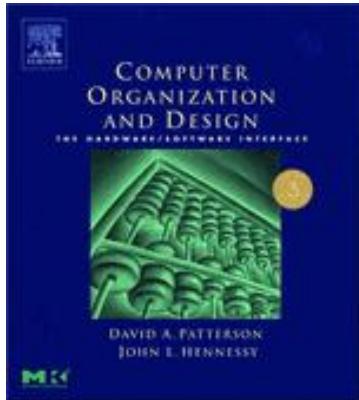
# Your final grade

---

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

# Texts

---



- 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

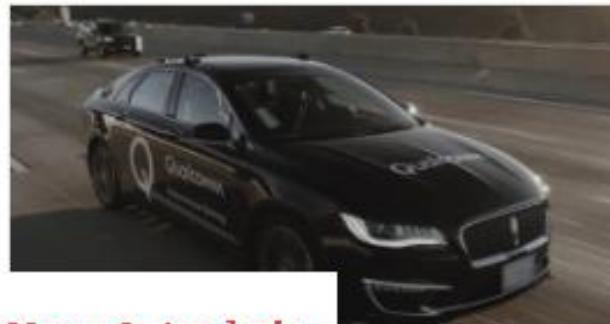
# Computer Architecture (计算机体系结构)

## Lecture #2 – Number Representation



Lecturer Yuanqing Cheng (成元庆)

[www.cadetlab.cn](http://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

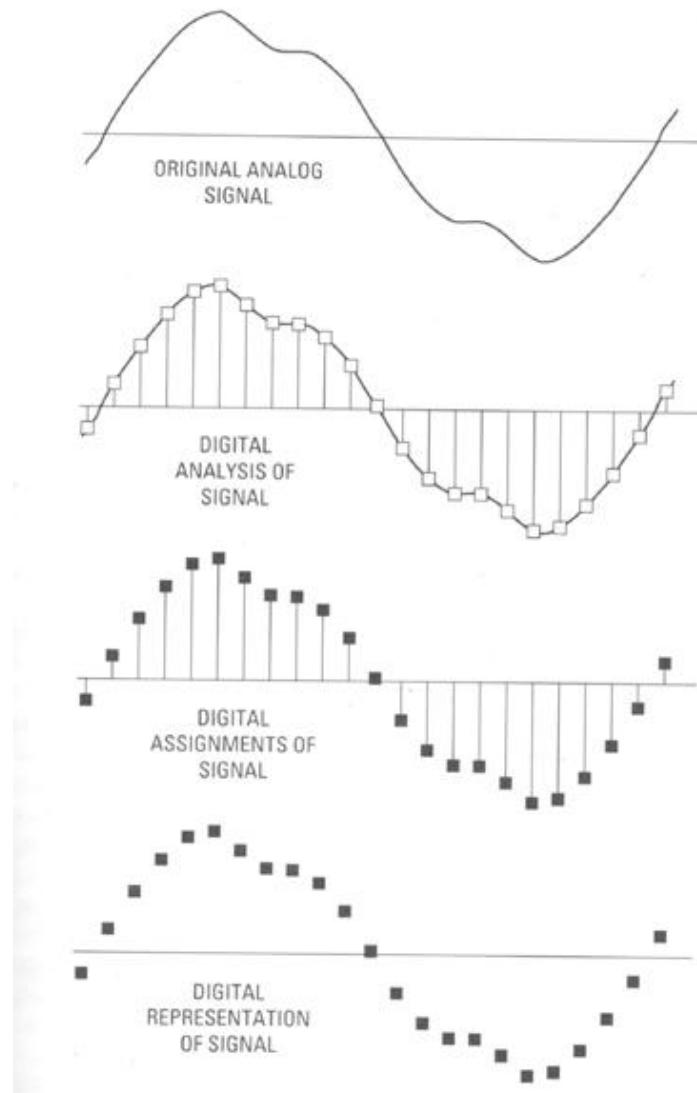
- Real world is analog!
- To import analog information, we must do two things

- Sample

- E.g., for a CD, every 44,100ths of a second, we ask a music signal how loud it is.

- Quantize

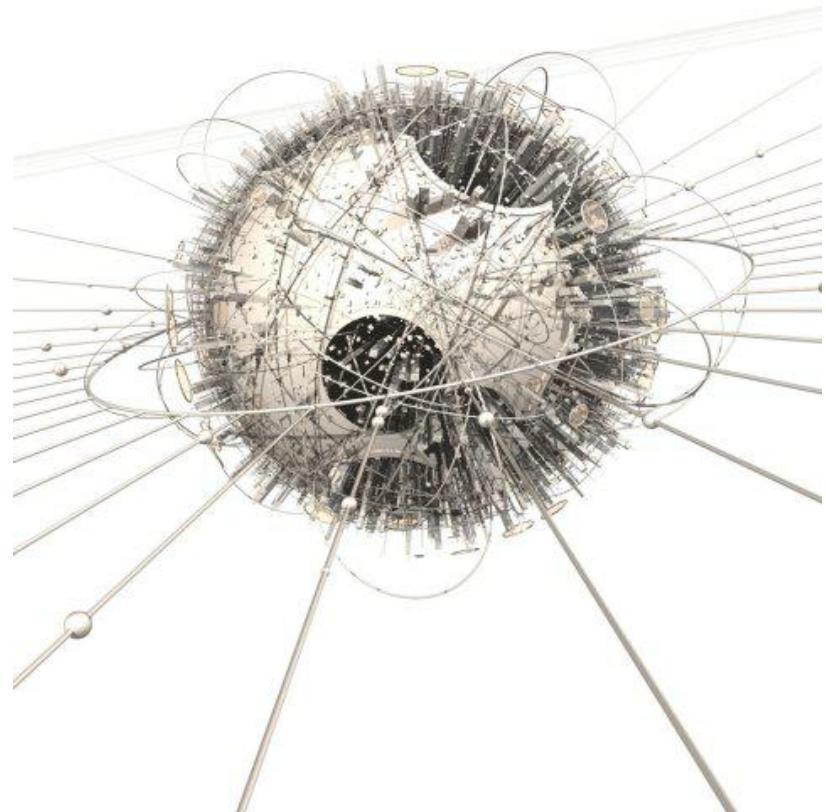
- For every one of these samples, we figure out where, on a 16-bit (65,536 tic-mark) “yardstick”, it lies.



[www.joshuadysart.com/journal/archives/digital\\_sampling.gif](http://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](http://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 $\pi$ ?

---



- 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)  $\infty$

# What to do with representations of numbers?

- Just what we do with numbers!

- Add them

1 1

- Subtract them

1 0 1 0

- Multiply them

+ 0 1 1 1

- Divide them

-----

- Compare them

1 0 0 0 1

- Example:  $10 + 7 = 17$

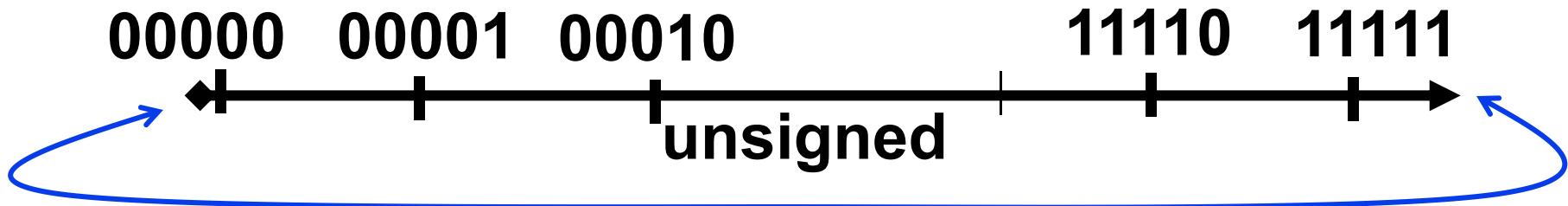
- ...so simple to add in binary that we can build circuits to do it!

- 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  $\infty$  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

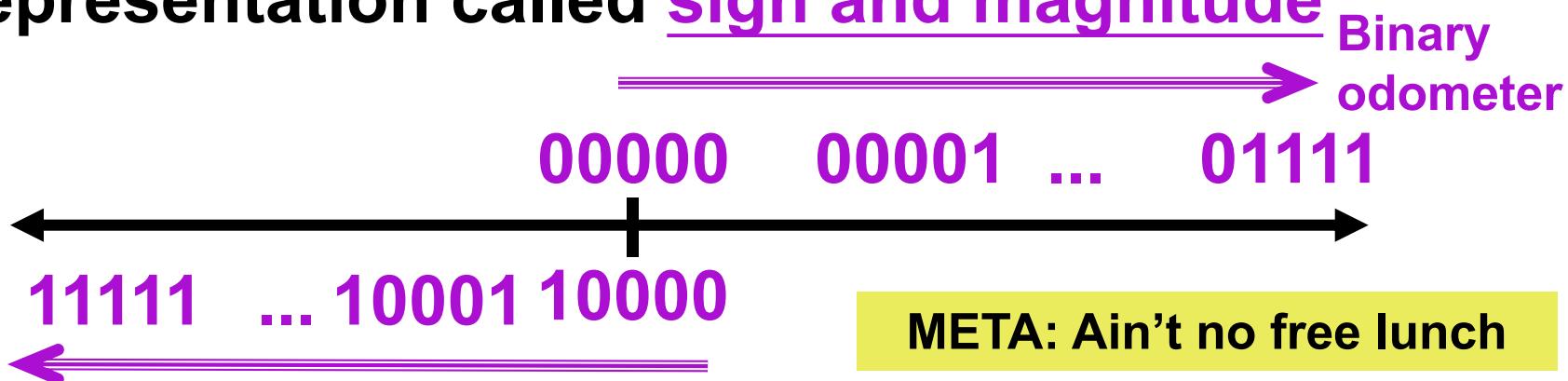


- Obvious solution: define leftmost bit to be sign!

- $0 \rightarrow +$        $1 \rightarrow -$

- Rest of bits can be numerical value of number

- Representation called sign and magnitude

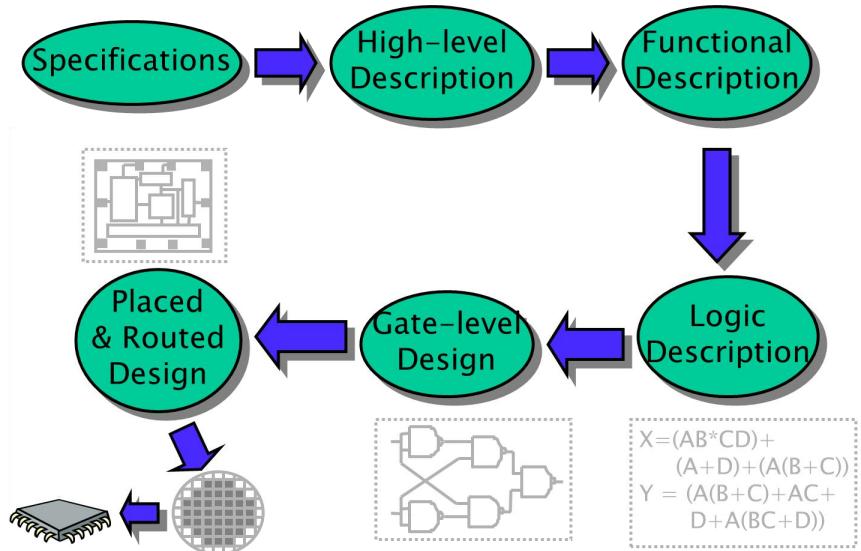


# Shortcomings of sign and magnitude?

- 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>

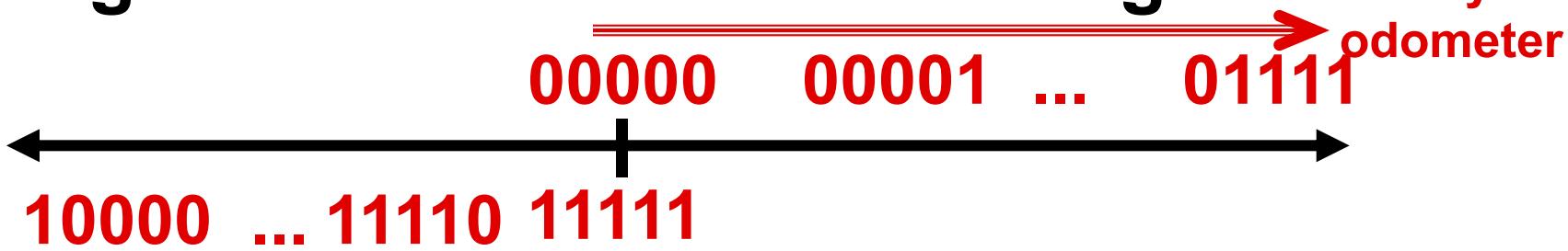


# Another try: complement the bits

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

- Called **One's Complement**

- Note: positive numbers have leading 0s, negative numbers have leadings 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
  - $0x00000000 = +0_{ten}$
  - $0xFFFFFFF = -0_{ten}$
- 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  $\geq 0$ , 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:

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

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

$$= 1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= -2^3 + 2^2 + 0 + 2^0$$

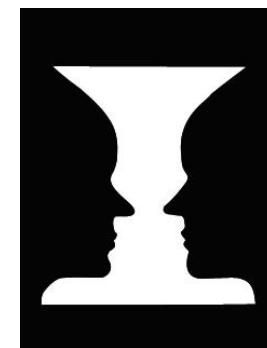
$$= -8 + 4 + 0 + 1$$

$$= -8 + 5$$

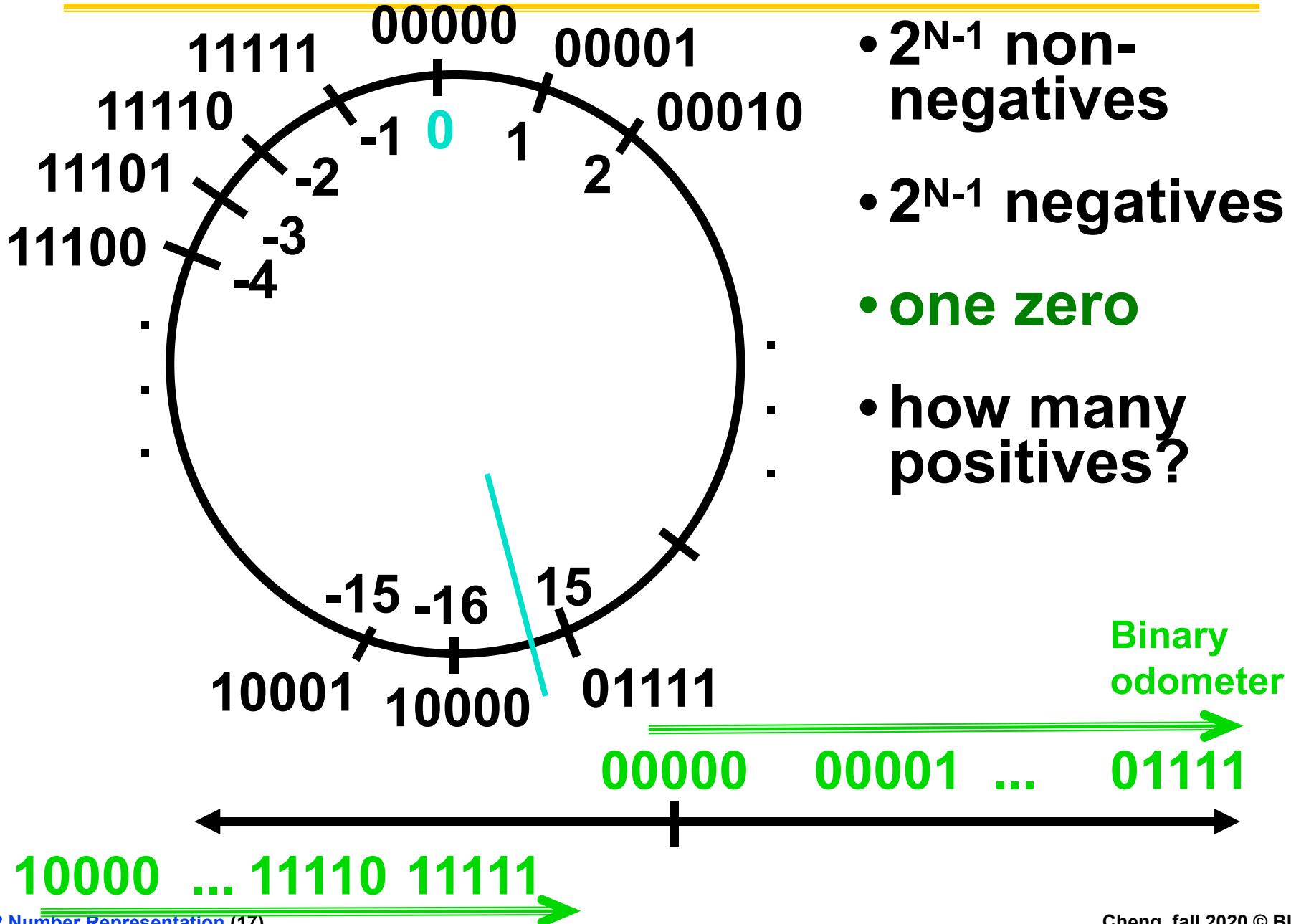
$$= -3_{\text{ten}}$$

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

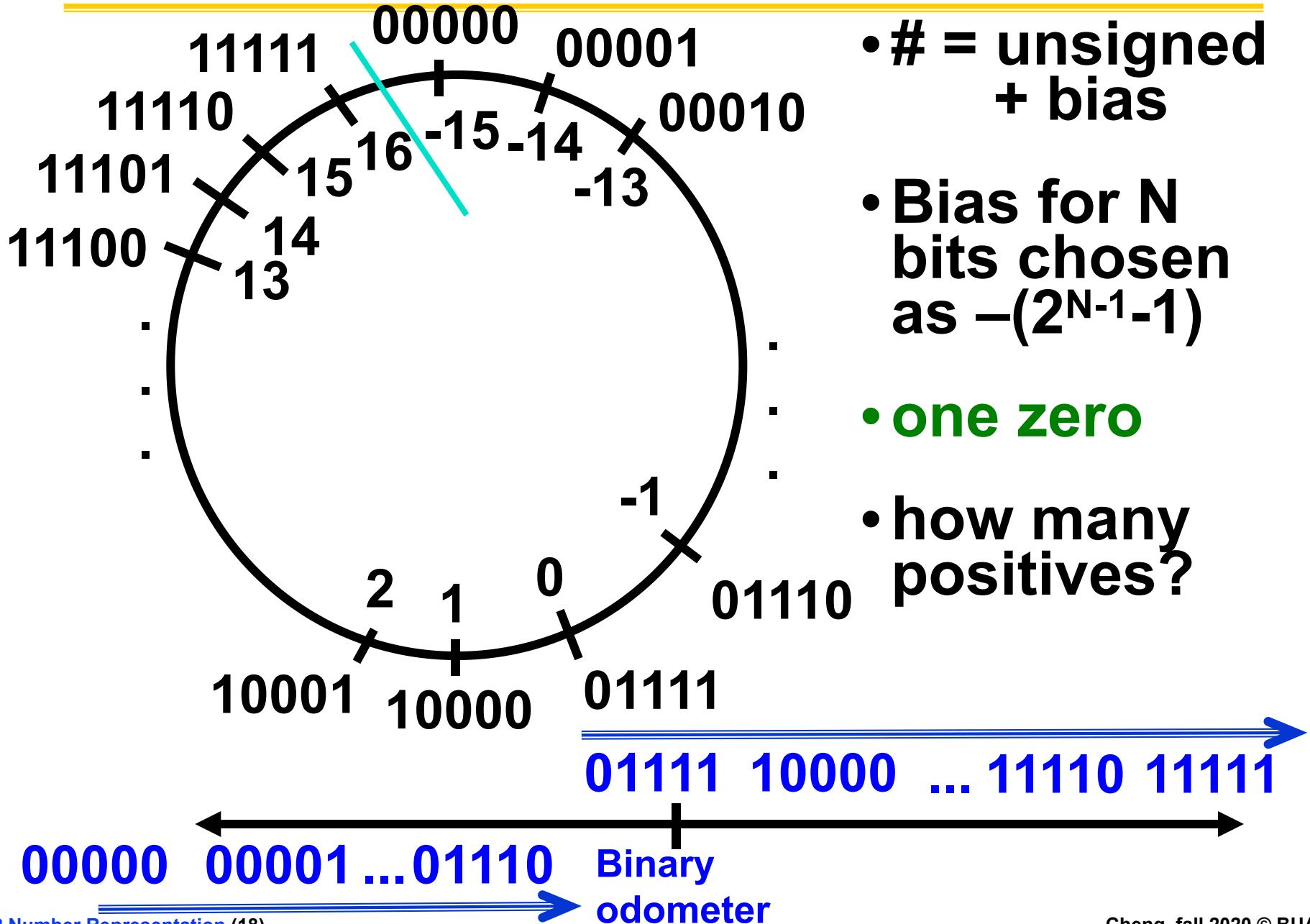
$$\begin{array}{ll} x: & 1101_{\text{two}} \\ x': & 0010_{\text{two}} \\ +1: & 0011_{\text{two}} \\ ()': & 1100_{\text{two}} \\ +1: & 1101_{\text{two}} \end{array}$$



## 2's Complement Number “line”: N = 5



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



# 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.,  $11_{10} = 1011.0_2 \rightarrow 10.110_2 = (11/4)_{10} = 2.75_{10}$**

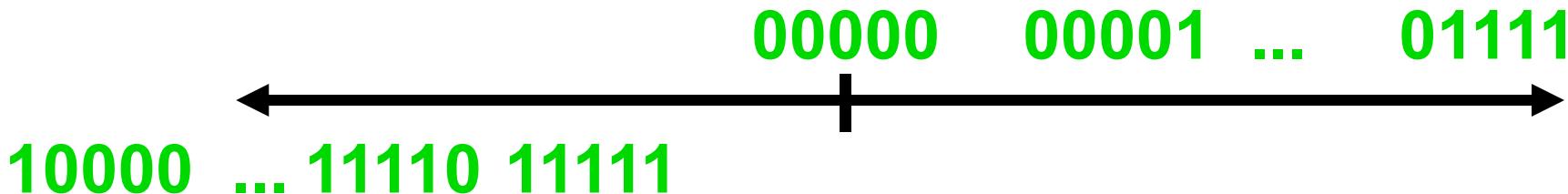
# And in summary...

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`) :



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



- **Overflow: numbers  $\infty$ ; 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 <sub>two</sub> =	0 <sub>ten</sub>
0000 ... 0000 0000 0000 0001 <sub>two</sub> =	1 <sub>ten</sub>
0000 ... 0000 0000 0000 0010 <sub>two</sub> =	2 <sub>ten</sub>

0111 ... 1111 1111 1111 1101 <sub>two</sub> =	2,147,483,645 <sub>ten</sub>
0111 ... 1111 1111 1111 1110 <sub>two</sub> =	2,147,483,646 <sub>ten</sub>
0111 ... 1111 1111 1111 1111 <sub>two</sub> =	2,147,483,647 <sub>ten</sub>
1000 ... 0000 0000 0000 0000 <sub>two</sub> =	-2,147,483,648 <sub>ten</sub>
1000 ... 0000 0000 0000 0001 <sub>two</sub> =	-2,147,483,647 <sub>ten</sub>
1000 ... 0000 0000 0000 0010 <sub>two</sub> =	-2,147,483,646 <sub>ten</sub>

1111 ... 1111 1111 1111 1101 <sub>two</sub> =	-3 <sub>ten</sub>
1111 ... 1111 1111 1111 1110 <sub>two</sub> =	-2 <sub>ten</sub>
1111 ... 1111 1111 1111 1111 <sub>two</sub> =	-1 <sub>ten</sub>

- One zero; 1st bit called sign bit
- 1 “extra” negative: no positive 2,147,483,648<sub>ten</sub>

## 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  $-4_{\text{ten}}$  to 32-bit:

1111 1111 1111 1100<sub>two</sub>

1111 1111 1111 1111 1111 1111 1111 1100<sub>two</sub>

## **Chapter 2 Week2: MIPS**

### **2.1 MIPS Arithmetic**

# Computer Architecture (计算机体系结构)

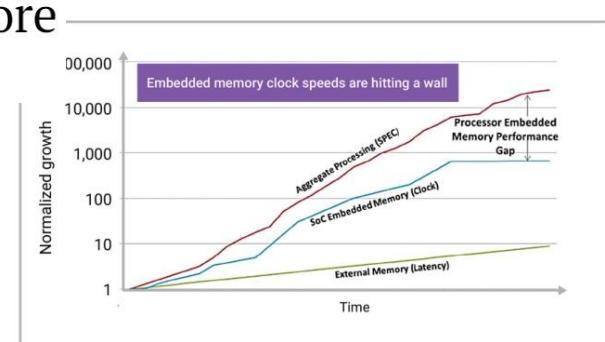
## Lecture #3 – Introduction to MIPS Assembly language : Arithmetic



Lecturer Yuanqing Cheng (成元庆)

[www.cadetlab.cn](http://www.cadetlab.cn)

Meeting Increasing Performance Requirements in Embedded Applications with Scalable Multicore Processors



- 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  $\infty$ ; computers finite, errors!

# 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, ...

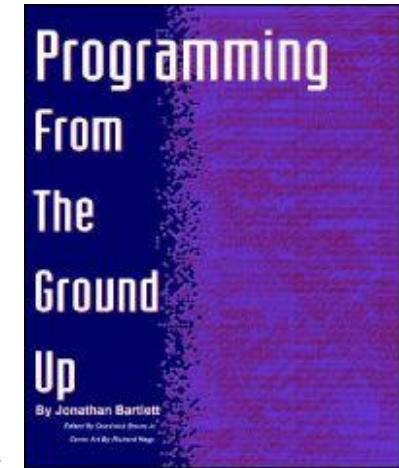
# Book: *Programming From the Ground Up*

---

*"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

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

# 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 => so no subi`
  - `addi $s0,$s1,-10 (in MIPS)`  
 $f = g - 10 \text{ (in C)}$   
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



# Computer Architecture (计算机体系结构)

## Lecture 4 – Introduction to MIPS Data Transfer & Decisions I

Lecturer  
Yuanqing  
Cheng

2020-09-11



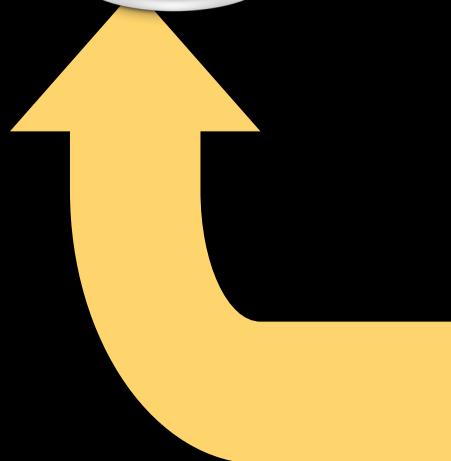
# Review

- 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**

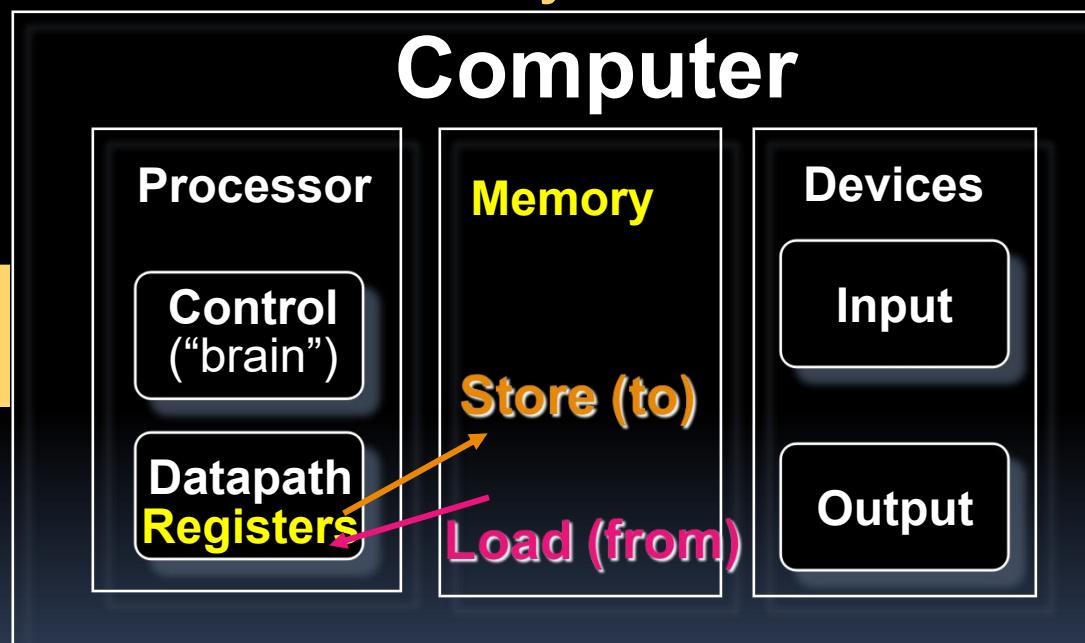
# Assembly Operands: Memory

- 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];`

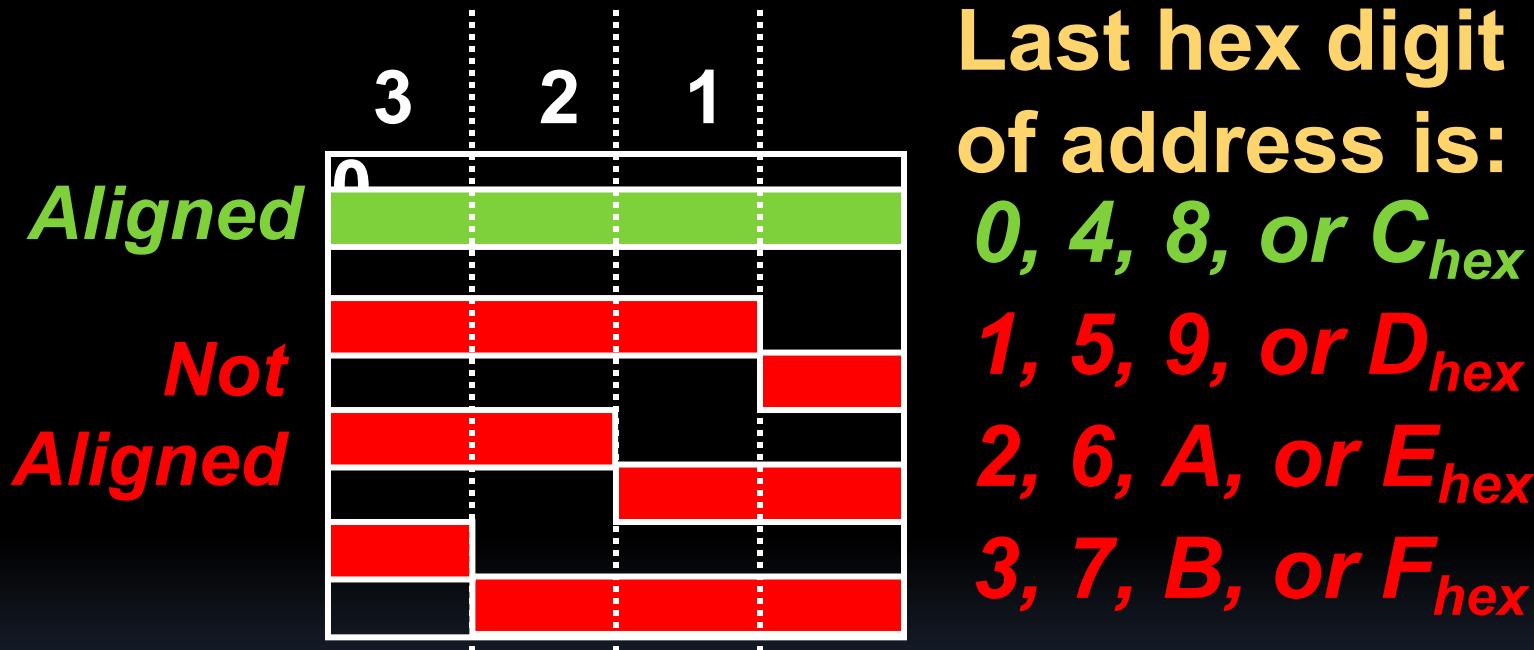
- `g: $s1, h: $s2, $s3: base address of A`
- 1st transfer from memory to register:  
`lw $t0, 20($s3) # $t0 gets A[5]`
  - Add 20 to `$s3` to select `A[5]`, put into `$t0`
- Next add it to `h` and place in `g`  
`add $s1, $s2, $t0 # $s1 = h+A[5]`

# 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



- 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:

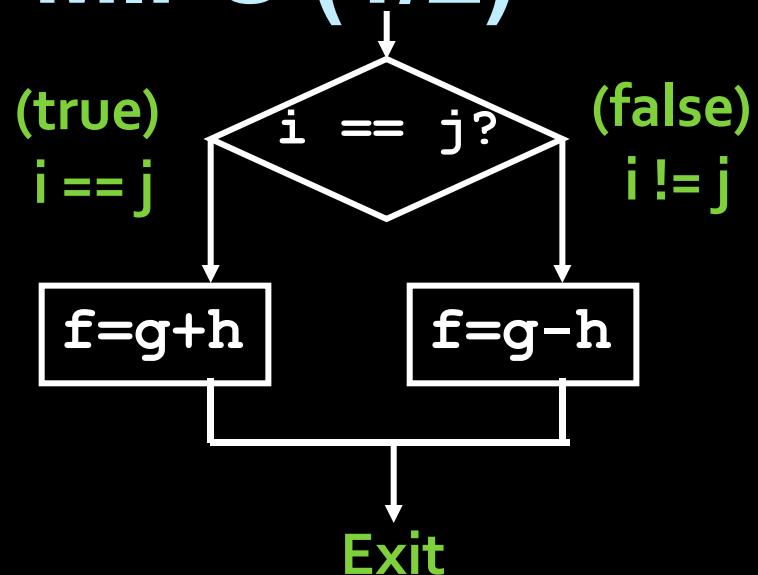
`j label`

- 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

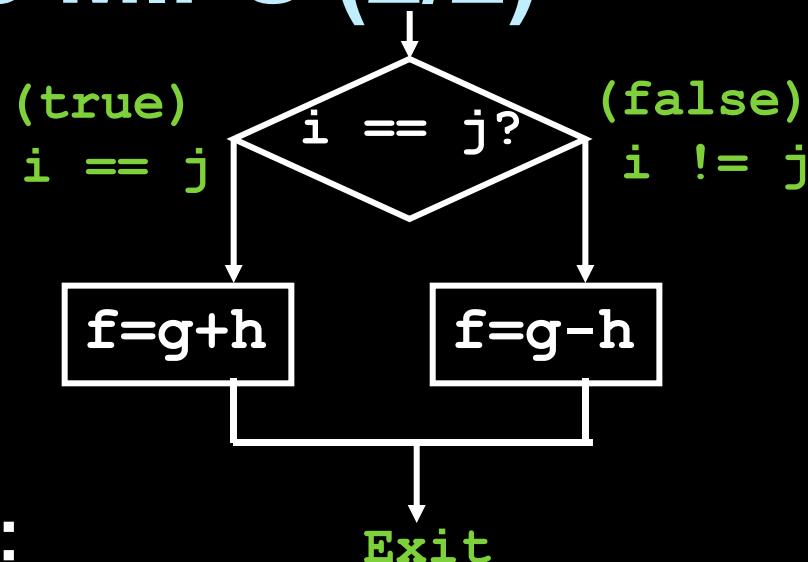
i: \$s3

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  
**Instruction**  
( $x, y$  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`

## 2.3 Addition to Data Transfer



# Computer Architecture (计算机体系结构)

## Lecture 5 Introduction to MIPS : Decisions II

Lecturer  
Yuanqing  
Cheng

2020-09-14

Germany Taking the Autobahn to  
Autonomy



# Review

---

- 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, so 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**

# Last time: Loading, Storing bytes 1/2

---

- In addition to word data transfers (**lw**, **sw**), MIPS has **byte** data transfers:
  - load byte: **lb**
  - store byte: **sb**
- same format as **lw**, **sw**
- E.g., **lb \$s0, 3(\$s1)**
  - *contents of memory location with address = sum of “3” + contents of register s1 is copied to the low byte position of register s0.*

# Loading, Storing bytes 2/2

- What do with other 24 bits in the 32 bit register?

- lb: sign extends to fill upper 24 bits



- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:
  - load byte unsigned: **lbu**

# Overflow in Arithmetic (1/2)

---

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

$$\begin{array}{r} 15 \\ + 3 \\ \hline 18 \end{array}$$

$$\begin{array}{r} 1111 \\ + 0011 \\ \hline 10010 \end{array}$$

- But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.

# Overflow in Arithmetic (2/2)

---

- Some languages detect overflow (Ada), some don't (C)
- MIPS solution is 2 kinds of arithmetic instructs:
  - These cause overflow to be detected
    - add (**add**)
    - add immediate (**addi**)
    - subtract (**sub**)
  - These do not cause overflow detection
    - add unsigned (**addu**)
    - add immediate unsigned (**addiu**)
    - subtract unsigned (**subu**)
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce **addu**, **addiu**, **subu**

# Two “Logic” Instructions

---

- Here are 2 more new instructions
- Shift Left: **sll \$s1,\$s2,2**  
**#s1=s2<<2**
  - Store in  $\$s1$  the value from  $\$s2$  shifted 2 bits to the left (they fall off end), inserting 0's on right;  $<<$  in C.
  - Before:  $0000\ 0002_{\text{hex}}$   
 $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}}$
  - After:  $0000\ 0008_{\text{hex}}$   
 $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{\text{two}}$
  - What arithmetic effect does shift left have?
- Shift Right: **srl** is opposite shift; **>>**

# Loops in C/Assembly (1/3)

---

- Simple loop in C; **A[]** is an array of ints

```
do { g = g + A[i];  
     i = i + j;  
 } while (i != h);
```

- Rewrite this as:

```
Loop: g = g + A[i];  
      i = i + j;  
      if (i != h) goto Loop;
```

- Use this mapping:

g, h, i, j, base of A  
\$s1, \$s2, \$s3, \$s4, \$s5

# Loops in C/Assembly (2/3)

---

- Final compiled MIPS code:

```
Loop: sll $t1,$s3,2      # $t1= 4*I  
      addu $t1,$t1,$s5    # $t1=addr A+4i  
      lw   $t1,0($t1)     # $t1=A[i]  
      addu $s1,$s1,$t1    # g=g+A[i]  
      addu $s3,$s3,$s4    # i=i+j  
      bne $s3,$s2,Loop    # goto Loop  
                      # if i!=h
```

- Original code:

```
Loop: g = g + A[i];  
      i = i + j;  
      if (i != h) goto Loop;
```

# Loops in C/Assembly (3/3)

---

- There are three types of loops in C:
  - **while**
  - **do... while**
  - **for**
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to these loops as well.
- Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision-making is conditional branch

# Inequalities in MIPS (1/4)

- Until now, we've only tested equalities (`==` and `!=` in C). General programs need to test `<` and `>` as well.
- Introduce MIPS Inequality Instruction:
  - “Set on Less Than”
  - Syntax: `slt reg1, reg2, reg3`
  - Meaning: `reg1 = (reg2 < reg3);`

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

Same thing...

“set” means “change to 1”,  
“reset” means “change to 0”.

# Inequalities in MIPS (2/4)

---

- How do we use this? Compile by hand:

```
if (g < h) goto Less; #g:$s0, h:$s1
```

- Answer: compiled MIPS code...

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h  
bne $t0,$0,Less # goto Less  
# if $t0!=0  
# (if (g<h) ) Less:
```

- Register \$0 always contains the value 0, so **bne** and **beq** often use it for comparison after an **slt** instruction.
- A **slt → bne** pair means **if (... < ...)** **goto...**

# Inequalities in MIPS (3/4)

---

- Now we can implement  $<$ ,  
but how do we implement  $>$ ,  $\leq$  and  $\geq$ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement  $\leq$  in one or more  
instructions using just **slt** and **branches**?
  - What about  $>?$
  - What about  $\geq?$

# Inequalities in MIPS (4/4)

---

```
# a:$s0, b:$s1
slt $t0,$s0,$s1 # $t0 = 1 if a<b
beq $t0,$0,skip # skip if a >= b
    <stuff>      # do if a<b
skip:
```

Two independent variations possible:

Use **slt \$t0,\$s1,\$s0** instead of  
**slt \$t0,\$s0,\$s1**

Use **bne** instead of **beq**

# Immediates in Inequalities

---

- There is also an immediate version of **slt** to test against constants: **slti**
  - Helpful in **for** loops

C           **if** (g >= 1) **goto** Loop

---

M

I        **slti** \$t0,\$s0,1                   **#** \$t0 = 1 if  
P    **#** \$s0<1 (g<1)

S        **beq**   \$t0,\$0,Loop                   **#** goto Loop  
  **#** if \$t0==0  
  **#** (if (g>=1))

An **slt** → **beq** pair means **if (... ≥ ...) goto...**

# What about unsigned numbers?

---

- Also **unsigned** inequality instructions:

**sltu, sltiu**

...which sets result to 1 or 0 depending on unsigned comparisons

- What is value of \$t0, \$t1?

(\$s0 = FFFF FFFA<sub>hex</sub>, \$s1 = 0000 FFFA<sub>hex</sub>)

**slt \$t0, \$s0, \$s1**

**sltu \$t1, \$s0, \$s1**

# MIPS Signed vs. Unsigned – diff

---

meanings!

■ MIPS terms Signed/Unsigned “overloaded”:

- Do/Don't sign extend
  - (**lb**, **lbu**)
- Do/Don't overflow
  - (**add**, **addi**, **sub**, **mult**, **div**)
  - (**addu**, **addiu**, **subu**, **multu**, **divu**)
- Do signed/unsigned compare
  - (**slt**, **slti/sltu**, **sltiu**)

# Peer Instruction

---

```
Loop: addi $s0,$s0,-1      # i = i - 1
      slti $t0,$s1,2        # $t0 = (j < 2)
      beq  $t0,$0 ,Loop     # goto Loop if $t0 == 0
      slt   $t0,$s1,$s0       # $t0 = (j < i)
      bne   $t0,$0 ,Loop     # goto Loop if $t0 != 0
```

(\$s0=i, \$s1=j)

What C code properly fills in  
the blank in loop below?

do {i--;} while(\_);

a	a	b	b	c	c	d	d	e	e	2	2	2	2	2	2	2	&&	&&	&&	i	i	i	i	i	i
										>	>	>	>	>	>	>									
										>	>	>	>	>	>	>									
										>	>	>	>	>	>	>									
										>	>	>	>	>	>	>									

# “And in conclusion...”

---

- To help the **conditional branches** make decisions concerning inequalities, we introduce: “Set on Less Than” called **slt, slti, sltu, sltiu**
- One can store and load (signed and unsigned) **bytes** as well as words with **lb, lbu**
- Unsigned add/sub **don't cause overflow**
- New MIPS Instructions:  
**sll, srl, lb, lbu**  
**slt, slti, sltu, sltiu**  
**addu, addiu, subu**

# Bonus Slides

# Example: The C Switch Statement

---

(1/3)

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3.

Compile this C code:

```
switch (k) {  
    case 0: f=i+j; break; /* k=0 */  
    case 1: f=g+h; break; /* k=1 */  
    case 2: f=g-h; break; /* k=2 */  
    case 3: f=i-j; break; /* k=3 */  
}
```

# Example: The C Switch Statement

---

(2/3)

This is complicated, so **simplify**.

- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if (k==0) f=i+j;  
else if (k==1) f=g+h;  
else if (k==2) f=g-h;  
else if (k==3) f=i-j;
```

- Use this mapping:

```
f:$s0, g:$s1, h:$s2,  
i:$s3, j:$s4, k:$s5
```

# Example: The C Switch Statement

---

▪ (3/3)

Final compiled MIPS code:

```
bne $s5,$0,L1      # branch k!=0
add $s0,$s3,$s4    #k==0 so f=i+j
j Exit            # end of case so Exit
L1: addi $t0,$s5,-1 # $t0=k-1
bne $t0,$0,L2      # branch k!=1
add $s0,$s1,$s2    #k==1 so f=g+h
j Exit            # end of case so Exit
L2: addi $t0,$s5,-2 # $t0=k-2
bne $t0,$0,L3      # branch k!=2
sub $s0,$s1,$s2    #k==2 so f=g-h
j Exit            # end of case so Exit
L3: addi $t0,$s5,-3 # $t0=k-3
bne $t0,$0,Exit    # branch k!=3
sub $s0,$s3,$s4    # k==3 so f=i-j
```

Exit:

## 2.4 Function



# Computer Architecture (计算机体系结构)

## Lecture 6 – Introduction to MIPS : Decisions II

Lecturer  
Yuanqing  
Cheng

2020-09-14

# Review

---

- In order to help the conditional branches make decisions concerning inequalities, we introduce a single instruction: “Set on Less Than” called **slt**, **slti**, **sltu**, **sltiu**
- One can store and load (signed and unsigned) **bytes** as well as words
- Unsigned add/sub **don't cause overflow**
- New MIPS Instructions:
  - sll**, **srl**, **lb**, **sb**
  - slt**, **slti**, **sltu**, **sltiu**
  - addu**, **addiu**, **subu**

# C functions

---

```
main() {  
    int i,j,k,m;  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}
```

What information must compiler/programmer keep track of?

```
/* really dumb mult function */  
  
int mult (int mcand, int mlier){  
    int product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier -1; }  
    return product;  
}
```

What instructions can accomplish this?

# Function Call Bookkeeping

---

- Registers play a major role in keeping track of information for function calls.
- Register conventions:
  - Return address      **\$ra**
  - Arguments            **\$a0, \$a1, \$a2, \$a3**
  - Return value          **\$v0, \$v1**
  - Local variables      **\$s0, \$s1, ... , \$s7**
- The stack is also used; more later.

# Instruction Support for Functions

(1/6)

```
sum(a,b); . . . /* a,b:$s0,$s1 */
```

```
}
```

```
C int sum(int x, int y) {  
    return x+y;  
}
```

---

address (shown in decimal)

M 1000

I 1004

P 1008

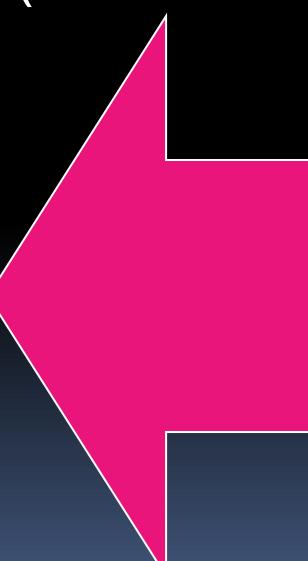
S 1012

1016

...

2000

2004



In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

# Instruction Support for Functions (2/6)

```
... sum(a,b); ... /* a,b:$s0,$s1 */  
}  
C int sum(int x, int y) {  
    return x+y;  
}
```

---

address (shown in decimal)

MIPS

```
1000 add $a0,$s0,$zero # x = a  
1004 add $a1,$s1,$zero # y = b  
1008 addi $ra,$zero,1016 #$ra=1016  
1012 j sum #jump to sum  
1016  
...  
2000 sum: add $v0,$a0,$a1  
2004 jr $ra # new instruction
```

# Instruction Support for Functions (3/6)

```
... sum(a,b); ... /* a,b:$s0,$s1 */  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

- Question: Why use **jr** here? Why not use **j**?
- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

M  
I  
P  
S



```
2000 sum: add $v0,$a0,$a1  
2004 jr $ra          # new instruction
```

# Instruction Support for Functions (4/6)

---

- Single instruction to jump and save return address: jump and link (**jal**)
- **Before:**

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum                      #goto sum
```
- **After:**

```
1008 jal sum    # $ra=1012, goto sum
```
- Why have a **jal**?
  - Make the common case fast: function calls very common.
  - Don't have to know where code is in memory with **jal**!

# Instruction Support for Functions (5/6)

---

- Syntax for `jal` (jump and link) is same as for `j` (jump):

`jal label`

- `jal` should really be called `laJ` for “link and jump”:
  - Step 1 (link): Save address of *next* instruction into `$ra`
    - Why next instruction? Why not current one?
  - Step 2 (jump): Jump to the given label

# Instruction Support for Functions (6/6)

---

- Syntax for **jr** (jump register):

**jr register**

- Instead of providing a label to jump to, the **jr** instruction provides a register which contains an address to jump to.
- Very useful for function calls:
  - **jal** stores return address in register (**\$ra**)
  - **jr \$ra** jumps back to that address

# Nested Procedures (1/2)

---

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

- Something called **sumSquare**, now **sumSquare** is calling **mult**.
- So there's a value in \$ra that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**.
- Need to save **sumSquare** return address before call to **mult**.

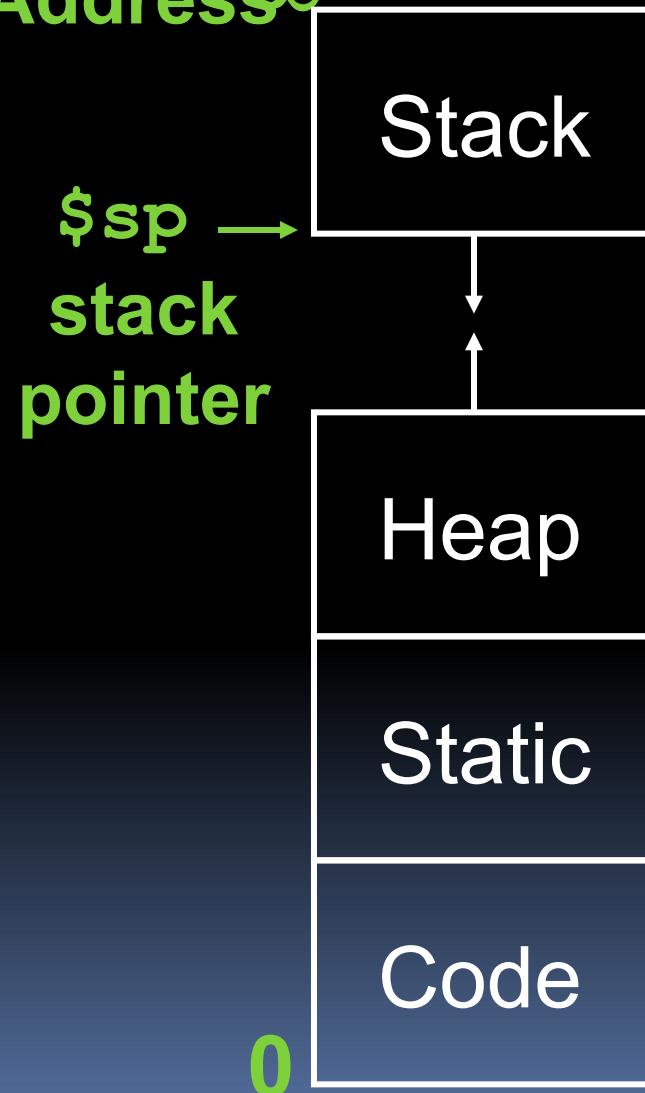
# Nested Procedures (2/2)

---

- In general, may need to save some other info in addition to `$ra`.
- When a C program is run, there are 3 important memory areas allocated:
  - **Static:** Variables declared once per program, cease to exist only after execution completes.  
E.g., C globals
  - **Heap:** Variables declared dynamically via `malloc`
  - **Stack:** Space to be used by procedure during execution; this is where we can save register values

# C memory Allocation review

Address<sup>∞</sup>



Space for saved procedure information

Explicitly created space,  
i.e., malloc ()

Variables declared once per program; e.g., globals

Program

# Using the Stack (1/2)

---

- So we have a register **\$sp** which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

# Using the Stack (2/2)

- Hand-compile `int sumSquare(int x, int y) {  
sumSquare:  
 return mult(x,x)+ y; }`

“push”

```
addi $sp,$sp,-8 # space on stack  
sw $ra, 4($sp) # save ret addr  
sw $a1, 0($sp) # save y  
add $a1,$a0,$zero # mult(x,x)  
jal mult # call mult  
lw $a1, 0($sp) # restore y  
add $v0,$v0,$a1 # mult() +y  
“pop” lw $ra, 4($sp) # get ret addr  
addi $sp,$sp,8 # restore stack  
jr $ra
```

**mult:** ...

# Steps for Making a Procedure Call

---

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. `jal` call
4. Restore values from stack.

# Rules for Procedures

---

- Called with a **jal** instruction,  
returns with a **jr \$ra**
- Accepts up to 4 arguments in  
**\$a0, \$a1, \$a2** and **\$a3**
- Return value is always in **\$v0**  
(and if necessary in **\$v1**)
- Must follow **register conventions**

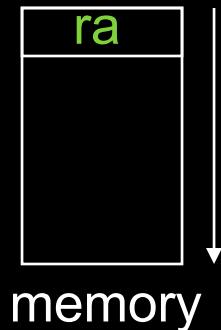
So what are they?

# Basic Structure of a Function

## Prologue

```
entry_label:  
addi $sp,$sp, -framesize  
sw $ra, framesize-4($sp) # save $ra  
save other regs if need be
```

**Body . . . (call other functions...)**



## Epilogue

```
restore other regs if need be  
lw $ra, framesize-4($sp) # restore $ra  
addi $sp,$sp, framesize  
jr $ra
```

# MIPS Registers

The constant 0	\$0	\$zero
Reserved for Assembler	\$1	\$at
Return Values	\$2-\$3	\$v0-
\$v1		
Arguments	\$4-\$7	
\$a0-\$a3		
Temporary		\$8-\$15
	\$t0-\$t7	
Saved		\$s0-
\$s7	\$16-\$23	
More Temporary	\$24-\$25	\$t8-\$t9
Used by Kernel	\$26-27	\$k0-
\$k1		
Global Pointer	\$28	\$gp
Stack Pointer	\$29	\$sp
Frame Pointer	\$30	\$fp
Return Address	\$31	
\$ra		

# Other Registers

---

- **\$at**: may be used by the assembler at any time; unsafe to use
- **\$k0-\$k1**: may be used by the OS at any time; unsafe to use
- **\$gp, \$fp**: don't worry about them
- Note: Feel free to read up on **\$gp** and **\$fp** in Appendix A, but you can write perfectly good MIPS code without them.

# Peer Instruction

---

```
int fact(int n) {  
    if(n == 0) return 1; else return(n*fact(n-1)); }
```

When translating this to MIPS...

- 1) We COULD copy \$a0 to \$a1 (& then not store \$a0 or \$a1 on the stack) to store n across recursive calls.
- 2) We MUST save \$a0 on the stack since it gets changed.
- 3) We MUST save \$ra on the stack since we need to know where to return to...

1	2	3
a)	FFF	
b)	FFT	
c)	FTF	
c)	FTT	
d)	TFF	
d)	TFT	
e)	TTF	
e)	TTT	

# “And in Conclusion...”

---

- Functions called with **jal**, return with **jr \$ra**.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far...
  - Arithmetic: **add**, **addi**, **sub**, **addu**, **addiu**, **subu**
  - Memory: **lw**, **sw**, **lb**, **sb**
  - Decision: **beq**, **bne**, **slt**, **slti**, **sltu**, **sltiu**
  - Unconditional Branches (Jumps): **j**, **jal**, **jr**
- Registers we know so far
  - All of them!

## 2.5 Logic



# Computer Architecture (计算机体系结构)

## Lecture 7 – Introduction to MIPS Procedures II & Logical Ops

Lecturer  
Yuanqing  
Cheng

2020-09-18

# Review

---

- Functions called with **jal**, return with **jr \$ra**.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far...
  - Arithmetic: **add**, **addi**, **sub**, **addu**, **addiu**, **subu**
  - Memory: **lw**, **sw**, **lb**, **sb**
  - Decision: **beq**, **bne**, **slt**, **slti**, **sltu**, **sltiu**
  - Unconditional Branches (Jumps): **j**, **jal**, **jr**
- Registers we know so far
  - All of them!
  - There are CONVENTIONS when calling procedures!

# Register Conventions (1/4)

---

- CalleR: the calling function
- CalleE: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.

# Register Conventions (2/4) – saved

---

- \$0: No Change. Always 0.
- \$s0-\$s7: Restore if you change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- \$sp: Restore if you change. The stack pointer must point to the same place before and after the `jal` call, or else the caller won't be able to restore values from the stack.
- HINT -- All saved registers start with S!

# Register Conventions (2/4) – volatile

---

- **\$ra: Can Change.** The `jal` call itself will change this register. Caller needs to save on stack if nested call.
- **\$v0-\$v1: Can Change.** These will contain the new returned values.
- **\$a0-\$a3: Can change.** These are volatile argument registers. Caller needs to save if they are needed after the call.
- **\$t0-\$t9: Can change.** That's why they're called temporary: any procedure may change them at any time. Caller needs to save if they'll need them afterwards.

# Register Conventions (4/4)

---

- What do these conventions mean?
  - If function **R** calls function **E**, then function **R** must save any temporary registers that it may be using onto the stack before making a `jal` call.
  - Function **E** must save any **S** (saved) registers it intends to use before garbling up their values
- Remember: caller/callee need to save only temporary/saved registers **they are using**, not all registers.

# Parents leaving for weekend analogy (1/5)

---

- Parents (`main`) leaving for weekend
- They (`caller`) give keys to the house to kid (`callee`) with the rules (`calling conventions`):
  - You can trash the temporary room(s), like the den and basement (`registers`) if you want, we don't care about it
  - BUT you'd better leave the rooms (`registers`) that we want to save for the guests untouched. “these rooms better look the same when we return!”
- Who hasn't heard this in their life?

# Parents leaving for weekend analogy (2/5)

---

- Kid now “owns” rooms (`registers`)
- Kid wants to use the `saved` rooms for a wild, wild party (`computation`)
- What does kid (`callee`) do?
  - Kid takes what was in these rooms and puts them in the garage (`memory`)
  - Kid throws the party, `trashes everything` (except garage, who ever goes in there?)
  - Kid restores the rooms the parents wanted `saved after the party` by `replacing the items from the garage` (`memory`) back into those `saved rooms`

# Parents leaving for weekend analogy (3/5)

---

- Same scenario, except before parents return and kid replaces **saved** rooms...
- Kid (**callee**) has left valuable stuff (**data**) all over.
  - Kid's friend (**anothercallee**) wants the house for a party **when the kid** is away
  - Kid knows that friend might **trash** the place destroying valuable stuff!
  - Kid remembers rule parents taught and now becomes the “heavy” (**caller**), instructing friend (**callee**) on good rules (**conventions**) of house.

# Parents leaving for weekend analogy (4/5)

---

- If kid had data in **temporary rooms** (which were going to be trashed), there are three options:
  - Move items directly to garage (**m em ory**)
  - Move items to **saved rooms** whose contents have already been moved to the garage (**m em ory**)
  - Optimize lifestyle (**code**) so that the amount you've got to move stuff back and forth from garage (**m em ory**) is minimized.
    - Mantra: “Minimize register footprint”
- Otherwise: “Dude, where's my data?!”

# Parents leaving for weekend analogy (5/5)

---

- Friend now “owns” rooms (registers)
- Friend wants to use the saved rooms for a wild, wild party (com putation)
- What does friend (callee) do?
  - Friend takes what was in these rooms and puts them in the garage (m em ory)
  - Friend throws the party, trashes everything (except garage)
  - Friend restores the rooms the kid wanted saved after the party by replacing the items from the garage (m em ory) back into those saved rooms

# Bitwise Operations

---

- So far, we've done arithmetic (**add**, **sub**, **addi**), mem access (**lw** and **sw**), & branches and jumps.
- All of these instructions view contents of register as a single quantity (e.g., signed or unsigned int)
- **New Perspective:** View register as 32 raw bits rather than as a single 32-bit number
  - Since registers are composed of 32 bits, wish to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions

# Logical Operators (1/3)

---

- Two basic logical operators:
  - AND: outputs 1 only if **all** inputs are 1
  - OR: outputs 1 if **at least one** input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

# Logical Operators (2/3)

---

- Logical Instruction Syntax:
  - 1 2,3,4
  - where
    - 1) operation name
    - 2) register that will receive value
    - 3) first operand (register)
    - 4) second operand (register) or immediate (numerical constant)
- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

# Logical Operators (3/3)

---

- Instruction Names:
  - **and, or**: Both of these expect the third argument to be a register
  - **andi, ori**: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all **bitwise**, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
  - C: Bitwise AND is **&** (e.g., ***z = x & y;***)
  - C: Bitwise OR is **|** (e.g., ***z = x | y;***)

# Uses for Logical Operators (1/3)

- Note that **anding** a bit with 0 produces a 0 at the output while **anding** a bit with 1 produces the original bit.
- This can be used to create a **mask**.
  - Example:

1011 0110 1010 0100 0011

**mask:** 0000 0000 0000 0000 0000

- The result of **anding** these:

0000 0000 0000 0000 0000

1101 1001 1010

1111 1111 1111

1101 1001 1010

**mask last 12 bits**

# Uses for Logical Operators (2/3)

---

- The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting to all 0s).
- Thus, the **and** operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in \$t0, then the following instruction would mask it:

**andi \$t0,\$t0,0xFFFF**

# Uses for Logical Operators (3/3)

---

- Similarly, note that **oring** a bit with 1 produces a 1 at the output while **oring** a bit with 0 produces the original bit.
- Often used to force certain bits to 1s.
  - For example, if \$t0 contains 0x12345678, then after this instruction:

```
ori $t0, $t0, 0xFFFF
```

... \$t0 will contain 0x1234~~FFFF~~FFF
    - (i.e., the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).

# Peer Instruction

---

```
r: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem  
...          ##### PUSH REGISTER(S) TO STACK?  
jal e      # Call e  
...          # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem  
jr $ra      # Return to caller of r  
  
e: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem  
jr $ra      # Return to r
```

What does **r** have to push on the stack before “**jal e**”?

- a) 1 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- b) 2 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- c) 3 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- d) 4 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- e) 5 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)

# Peer Instruction Answer

```
r: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem  
...          ##### PUSH REGISTER(S) TO STACK?  
jal e      # Call e  
...          # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem  
jr $ra      # Return to caller of r  
  
e: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem  
jr $ra      # Return to r
```

What does **r** have to push on the stack before “**jal e**”?

Saved      Volatile! -- need to push

- a) 1 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- b) 2 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- c) 3 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- d) 4 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- e) 5 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)



# “And in Conclusion...”

---

- **Register Conventions:** Each register has a purpose and limits to its usage. Learn these and follow them, even if you’re writing all the code yourself.
- **Logical and Shift Instructions**
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, **sll**, for multiplication by powers of 2
  - Use shift right logical, **srl**, for division by powers of 2 of unsigned numbers (**unsigned int**)
  - Use shift right arithmetic, **sra**, for division by powers of 2 of signed numbers (**int**)
- **New Instructions:**  
**and, andi, or, ori, sll, srl, sra**

# Bonus slides

---

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus

# Shift Instructions (review) (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift left by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0011 0100 0101 0110 0111 1000 0000 0000

# Shift Instructions (2/4)

---

- Shift Instruction Syntax:

1 2,3,4

...where

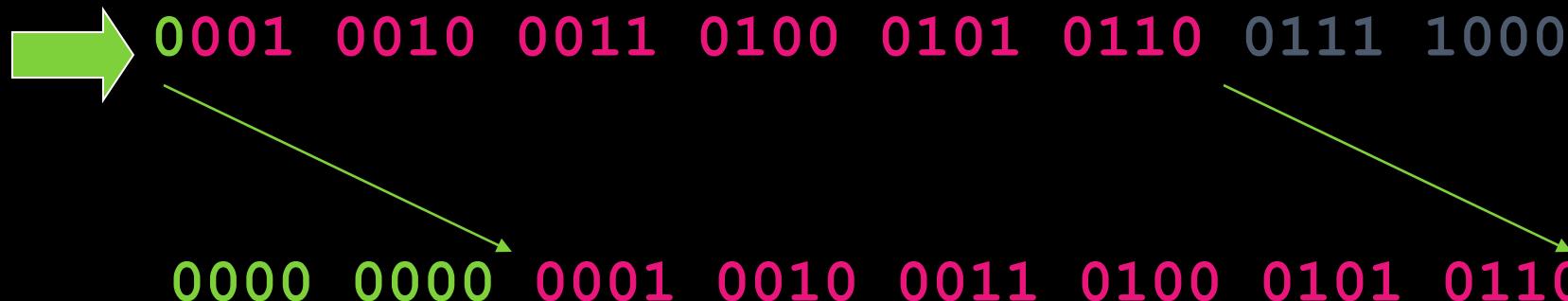
- 1) operation name
- 2) register that will receive value
- 3) first operand (register)
- 4) shift amount (constant < 32)

- MIPS shift instructions:

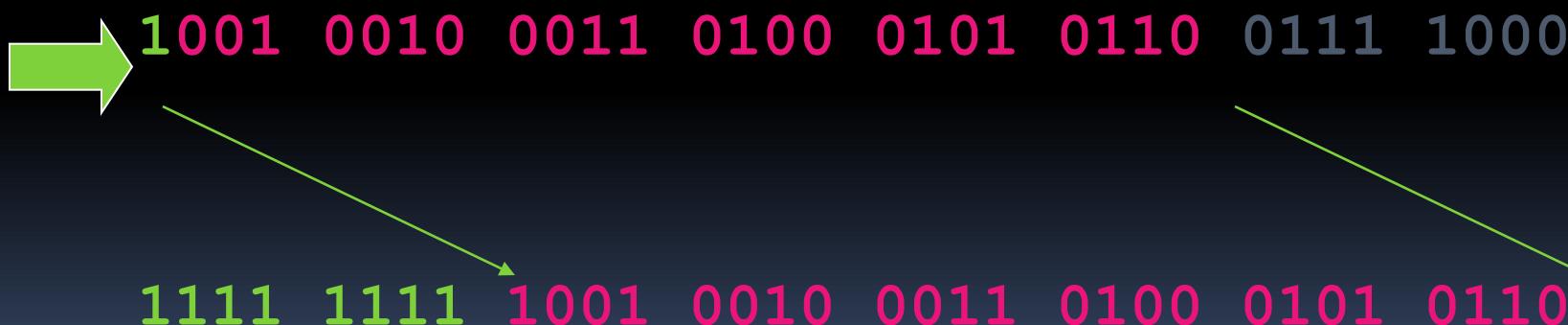
1. **sll** (shift left logical): shifts left and fills emptied bits with 0s
2. **srl** (shift right logical): shifts right and fills emptied bits with 0s
3. **sra** (shift right arithmetic): shifts right and fills emptied bits by sign extending

# Shift Instructions (3/4)

- Example: shift right arithmetic by 8 bits



- Example: shift right arithmetic by 8 bits



# Shift Instructions (4/4)

---

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:  
`a *= 8;` (in C)  
would compile to:  
`sll $s0,$s0,3` (in MIPS)
- Likewise, shift right to divide by powers of 2 (rounds towards  $-\infty$  )
  - remember to use `sra`

# Example: Fibonacci Numbers 1/8

---

- The Fibonacci numbers are defined as follows:  $F(n) = F(n - 1) + F(n - 2)$ ,  $F(0)$  and  $F(1)$  are defined to be 1
- In scheme, this could be written:

```
(define (Fib n)          (cond      (= n 0) 1)
      (= n 1) 1)
      (else (+ (Fib (-n 1))
                (Fib (-n 2))))))
```

# Example: Fibonacci Numbers 2/8

---

- Rewriting this in C we have:

```
int fib (int n) {  
    if(n == 0) {  
        return 1; }  
    if(n == 1) { return 1; }  
    return (fib (n - 1) + fib (n - 2));  
}
```

# Example: Fibonacci Numbers 3/8

---

- Now, let's translate this to MIPS!
- You will need space for three words on the stack
- The function will use one \$s register, \$s0
- Write the Prologue:

fib:

```
addi $sp, $sp, -12    # Space for three words
sw $ra, 8($sp)         # Save return address
sw $s0, 4($sp)         # Save s0
```

# Example: Fibonacci Numbers 4/8

---

- ° Now write the Epilogue:

fin:

```
lw $s0, 4($sp)      # Restore $s0
lw $ra, 8($sp)      # Restore return address
addi $sp, $sp, 12    # Pop the stack frame
jr $ra               # Return to caller
```



# Example: Fibonacci Numbers 5/8

---

- Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {  
    if(n == 0) { return 1; } /*Translate Me!*/ if(n == 1)  
    { return 1; } /*Translate Me!*/ return (fib(n - 1) + fib(n -  
2));  
}  
  
addi $v0, $zero, 1          # $v0 = 1  
beq $a0, $zero, fin        #  
addi $t0, $zero, 1          # $t0 = 1  
beq $a0, $t0, fin          #
```

Continued on next slide. . .



# Example: Fibonacci Numbers 6/8

---

- ° Almost there, but be careful, this part is tricky!

```
int fib (int n) {  
    ...  
    return (fib (n -1) + fib (n -2));  
}
```

```
addi $a0, $a0, -1          # $a0 = n - 1  
sw $a0, 0($sp)              # Need $a0 after jal  
jal fib                     # fib(n - 1)  
lw $a0, 0($sp)              # restore $a0  
addi $a0, $a0, -1          # $a0 = n - 2
```



# Example: Fibonacci Numbers 7/8

---

- ° Remember that \$vo is caller saved!

```
int fib (int n) {  
    . . .  
    return (fib (n -1) + fib (n -2));  
}
```

```
add $s0, $v0, $zero      # Place fib(n - 1)  
                          # somewhere it won't get  
                          # clobbered  
jal fib                  # fib(n - 2)  
add $v0, $v0, $s0          # $v0 = fib(n-1) + fib(n-2)
```

To the epilogue and beyond. . . .



# Example: Fibonacci Numbers 8/8

---

- ° Here's the complete code for reference:

```
fib: addi $sp, $sp, -12
      sw $ra, 8($sp)
      sw $s0, 4($sp)
      addi $v0, $zero, 1
      beq $a0, $zero, fin
      addi $t0, $zero, 1
      beq $a0, $t0, fin
      addi $a0, $a0, -1
      sw $a0, 0($sp)
      jal fib
```

```
           lw $a0, 0($sp)
           addi $a0, $a0, -1
           add $s0, $v0, $zero
           jal fib
           add $v0, $v0, $s0
           fin: lw $s0, 4($sp)
                  lw $ra, 8($sp)
                  addi $sp, $sp, 12
                  jr $ra
```



# Bonus Example: Compile This (1/5)

---

```
main() {  
    int i,j,k,m; /* i-m:$s0-$s3 */  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}  
  
int mult (int mcand, int mlier) {  
    int product;  
  
    product = 0;  
    while (mlier > 0) {  
        product += mcand;  
        mlier -= 1; }  
    return product;  
}
```

# Bonus Example: Compile This (2/5)

---

\_\_start:

...

```
add $a0,$s1,$0      # arg0 = j
add $a1,$s2,$0      # arg1 = k
jal mult            # call mult
add $s0,$v0,$0      # i = mult()
```

```
add $a0,$s0,$0      # arg0 = i
add $a1,$s0,$0      # arg1 = i
jal mult            # call mult
add $s3,$v0,$0      # m = mult()
```

...

```
j __exit    main() {
              int i,j,k,m; /* i-m:$s0-$s3 */
              ...
              i = mult(j,k); ...
              m = mult(i,i); ... }
```

# Bonus Example: Compile This (3/5)

---

- Notes:
  - **main** function ends with a jump to **\_\_exit**, not **jr \$ra**, so there's no need to save **\$ra** onto stack
  - all variables used in **main** function are saved registers, so there's no need to save these onto stack

# Bonus Example: Compile This (4/5)

---

**mult:**

**Loop:**

add	\$t0,\$0,\$0	# <i>prod=0</i>
slt	\$t1,\$0,\$a1	# <i>mlr &gt; 0?</i>
beq	\$t1,\$0,Fin	# <i>no=&gt;Fin</i>
add	\$t0,\$t0,\$a0	# <i>prod+=mc</i>
addi	\$a1,\$a1,-1	# <i>mlr-=1</i>
j	Loop	# <i>goto Loop</i>

**Fin:**

add	\$v0,\$t0,\$0	# <i>\$v0=prod</i>
jr	\$ra	# <i>return</i>

```
int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0)  {
        product += mcand;
        mlier -= 1; }
    return product;
}
```

# Bonus Example: Compile This

---

(5/5)

## Notes:

- no `jal` calls are made from `mult` and we don't use any saved registers, so we don't need to save anything onto stack
- temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller's on the stack.)
- `$a1` is modified directly (instead of copying into a temp register) since we are free to change it
- result is put into `$v0` before returning (could also have modified `$v0` directly)

## 2.6 Representation



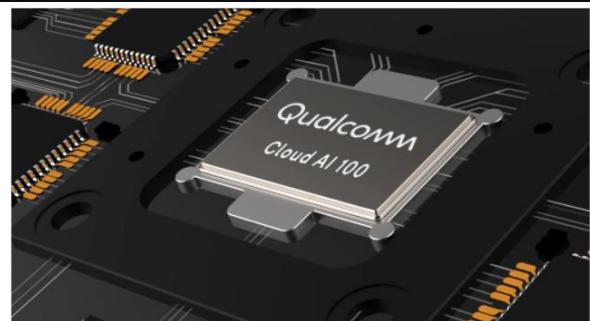
# Computer Architecture (计算机体系结构)

## Lecture 8 MIPS Instruction Representation I

Lecturer  
Yuanqing  
Chen

2020-09-18

Qualcomm Cloud AI 100 Promises Impressive Performance per Watt for Near-Edge AI



Qualcomm's Cloud AI 100 is targeted at near-edge applications including enterprise data centers and 5G infrastructure (Image: Qualcomm)

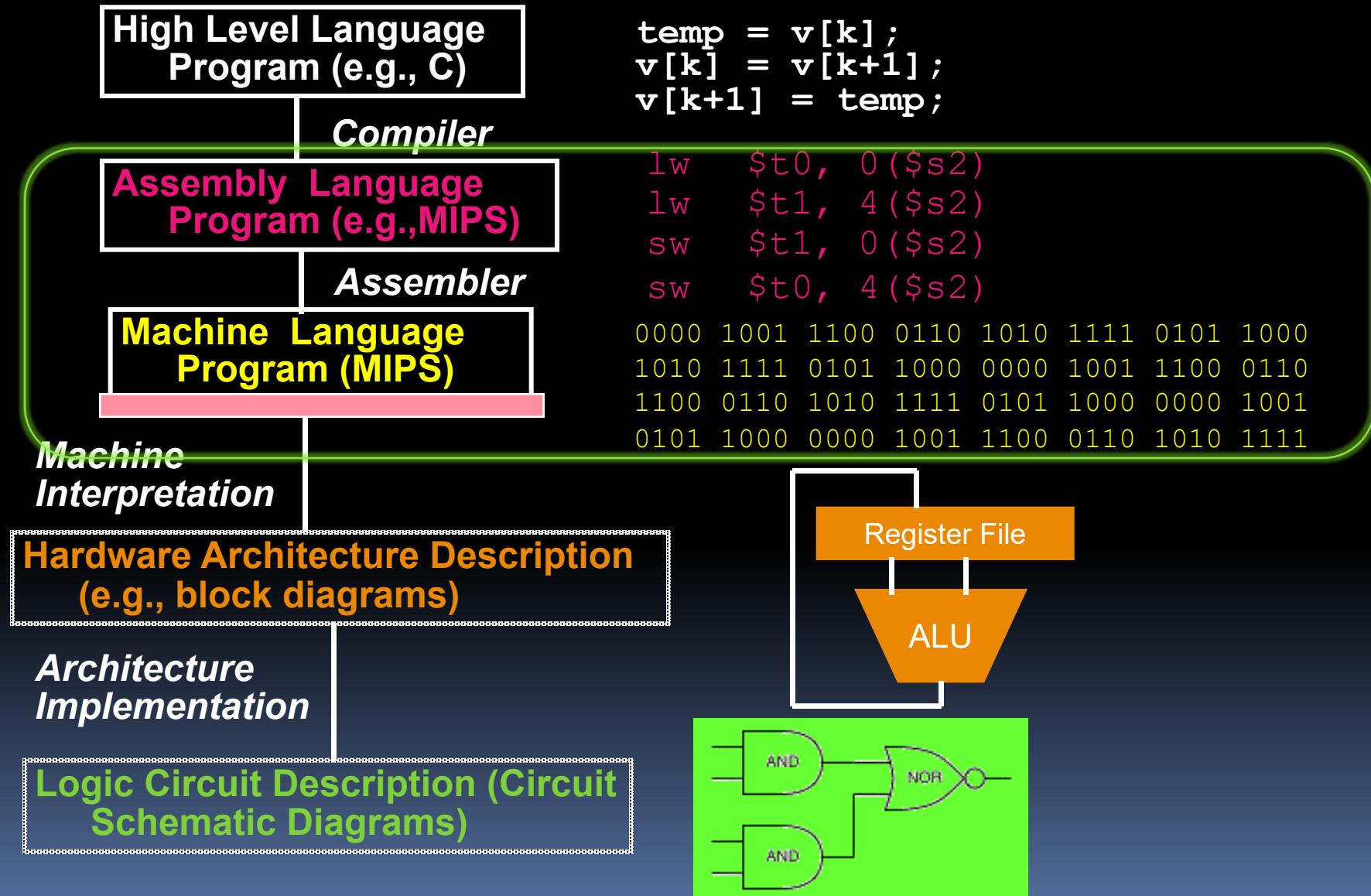


# Review

---

- **Register Conventions:** Each register has a purpose and limits to its usage. Learn these and follow them, even if you're writing all the code yourself.
- **Logical and Shift Instructions**
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, **sll**, for multiplication by powers of 2
  - Use shift right logical, **srl**, for division by powers of 2 of unsigned numbers (**unsigned int**)
  - Use shift right arithmetic, **sra**, for division by powers of 2 of signed numbers (**int**)
- **New Instructions:**  
**and, andi, or, ori, sll, srl, sra**

# Levels of Representation (abstractions)



# Overview – Instruction

---

## Representation

Big Idea: stored program

- consequences of stored program
- Instructions as numbers
- Instruction encoding
- MIPS instruction format for Add instructions
- MIPS instruction format for Immediate, Data transfer instructions

# Big Idea: Stored-Program Concept

---

- Computers built on 2 key principles:
  - Instructions are represented as bit patterns - can think of these as numbers.
  - Therefore, entire programs can be stored in memory to be read or written just like data.
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

# Consequence #1: Everything

---

## Addressed

Since all instructions and data are stored in memory, everything has a memory address: instructions, data words

- both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: **“Program Counter” (PC)**
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

# Consequence #2: Binary

---

## Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintoshes and PCs
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to “backward compatible” instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

# Instructions as Numbers (1/2)

---

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - **lw** and **sw** both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so “**add \$t0,\$0,\$0**” is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions be words too

# Instructions as Numbers (2/2)

---

- One word is 32 bits, so divide instruction word into “**fields**”.
- Each field tells processor something about instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format

# Instruction Formats

---

- **I-format**: used for instructions with immediates, **lw** and **sw** (since offset counts as an immediate), and branches (**beq** and **bne**),
  - (but not the shift instructions; later)
- **J-format**: used for **j** and **jal**
- **R-format**: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way.

# R-Format Instructions (1/5)

- Define “**fields**” of the following number of bits each:  $6 + 5 + 5 + 5 + 5 + 6 = 32$

6	5	5	5	5	6
---	---	---	---	---	---

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- Important:** On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.

# R-Format Instructions (2/5)

---

- What do these field integer values tell us?
  - `opcode`: partially specifies what instruction it is
    - Note: This number is equal to 0 for all R-Format instructions.
  - `funct`: combined with `opcode`, this number exactly specifies the instruction
- Question: Why aren't `opcode` and `funct` a single 12-bit field?
  - We'll answer this later.

# R-Format Instructions (3/5)

---

- More fields:
  - rs (Source Register): *generally* used to specify register containing first operand
  - rt (Target Register): *generally* used to specify register containing second operand (note that name is misleading)
  - rd (Destination Register): *generally* used to specify register which will receive result of computation

# R-Format Instructions (4/5)

---

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  - The word “generally” was used because there are exceptions that we’ll see later. E.g.,
    - **mult** and **div** have nothing important in the **rd** field since the dest registers are **hi** and **lo**
    - **mfhi** and **mflo** have nothing important in the **rs** and **rt** fields since the source is determined by the instruction (see COD)

# R-Format Instructions (5/5)

---

- Final field:
  - shamt: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).
  - This field is set to 0 in all but the shift instructions.
- For a detailed description of field usage for each instruction, see green insert in COD (You can bring with you to all exams)

# R-Format Example (1/2)

---

- MIPS Instruction:

**add \$8 , \$9 , \$10**

opcode = 0 (look up in table in book)

funct = 32 (look up in table in book)

rd = 8 (destination)

rs = 9 (first *operand*)

rt = 10 (second *operand*)

shamt = 0 (not a shift)

# R-Format Example (2/2)

- MIPS Instruction:

add \$8,\$9,\$10

Decimal number per field representation:

0	9	10	8	0	32
---	---	----	---	---	----

Binary number per field representation:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

hex

hex representation:

012A 4020<sub>hex</sub>

decimal representation:

19, 546, 144<sub>ten</sub>

Called a Machine Language Instruction

# Administrivia

---

- Remember to look at Appendix A (also on SPIM website), for MIPS assembly language details, including “assembly directives”, etc.

# I-Format Instructions (1/4)

---

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

# I-Format Instructions (2/4)

---

- Define “fields” of the following number of bits each:  $6 + 5 + 5 + 16 = 32$  bits



- Again, each field has a name:



- Key Concept:** Only one field is inconsistent with R-format. Most importantly, opcode is still in same location.

# I-Format Instructions (3/4)

---

- What do these fields mean?
  - `opcode`: same as before except that, since there's no `funct` field, `opcode` uniquely specifies an instruction in I-format
  - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent as possible with other formats while leaving as much space as possible for immediate field.
  - `rs`: specifies a register operand (if there is one)
  - `rt`: specifies register which will receive result of computation (this is why it's called the *target* register "rt") or other operand for some instructions.

# I-Format Instructions (4/4)

---

- The Immediate Field:
  - **addi, slti, sltiu**, the immediate is sign-extended to 32 bits. Thus, it's treated as a signed integer.
  - 16 bits → can be used to represent immediate up to  $2^{16}$  different values
  - This is large enough to handle the offset in a typical **lw** or **sw**, plus a vast majority of values that will be used in the **slti** instruction.
  - We'll see what to do when the number is too big in our next lecture...

# I-Format Example (1/2)

---

- MIPS Instruction:

**addi \$21,\$22,-50**

opcode = 8 (look up in table in book)

rs = 22 (register containing operand)

rt = 21 (target register)

immediate = -50 (by default, this is decimal)

# I-Format Example (2/2)

- MIPS Instruction:

**addi \$21,\$22,-50**

**Decimal/field representation:**

8	22	21	-50
---	----	----	-----

**Binary/field representation:**

001000	10110	10101	1111111111001110
--------	-------	-------	------------------

**hexadecimal representation:** 22D5

~~decimal representation:~~

**584,449,998<sub>ten</sub>**

# Peer Instruction

Which instruction has same representation as  $35_{\text{ten}}$ ?

- a) add \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct

- b) subu \$s0,\$s0,\$s0

opcode	rs	rt	rd	shamt	funct

- c) lw \$0, 0(\$0)

opcode	rs	rt	offset		

- d) addi \$0, \$0, 35

opcode	rs	rt	immediate		

- e) subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct

Registers numbers and names:

0: \$0, .. 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Opcodes and function fields (if necessary)

**add**: opcode = 0, funct = 32

**subu**: opcode = 0, funct = 35

**addi**: opcode = 8

**lw**: opcode = 35

# Peer Instruction Answer

Which instruction has same representation as  $35_{\text{ten}}$ ?

a) add \$0, \$0, \$0

0	0	0	0	0	32
---	---	---	---	---	----

b) subu \$s0,\$s0,\$s0

0	16	16	16	0	35
---	----	----	----	---	----

c) lw \$0, 0(\$0)

35	0	0			0
----	---	---	--	--	---

d) addi \$0, \$0, 35

8	0	0			35
---	---	---	--	--	----

e) subu \$0, \$0, \$0

0	0	0	0	0	35
---	---	---	---	---	----

Registers numbers and names:

0: \$0, .. 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Opcodes and function fields (if necessary)

**add**: opcode = 0, funct = 32

**subu**: opcode = 0, funct = 35

**addi**: opcode = 8

**lw**: opcode = 35

# In conclusion...

---

- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use lw and sw).
- Computer actually stores programs as a series of these 32-bit numbers.
- **MIPS Machine Language Instruction:** 32 bits representing a single instruction

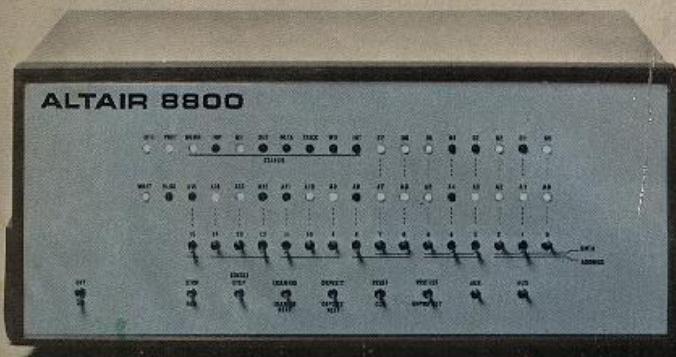
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt		immediate	



EXCLUSIVE!

## ALTAIR 8800

**The most powerful minicomputer project ever presented—can be built for under \$400**



BY H. EDWARD ROBERTS AND WILLIAM YATES

THE era of the computer in every home—a favorite topic among science-fiction writers—has arrived! It's made possible by the POPULAR ELECTRONICS/MITS Altair 8800, a full-blown computer that can hold its own against sophisticated minicomputers now on the market. And it doesn't cost several thousand dollars. In fact, it's in a color TV-receiver's price class—under \$400 for a complete kit.

The Altair 8800 is not a "demonstrator" or souped-up calculator. It is the most powerful computer ever presented as a construction project in any electronics magazine. In many ways, it represents a revolutionary development in electronic design and thinking.

The Altair 8800 is a parallel 8-bit word/16-bit address computer with an instruction cycle time of 2  $\mu$ s. Its central

processing unit is a new LSI chip that is many times more powerful than previous IC processors. It can accommodate 256 inputs and 256 outputs, all directly addressable, and has 78 basic machine instructions (as compared with 40 in the usual minicomputer). This means that you can write an extensive and detailed program. The basic computer has 256 words of memory, but it can be economically expanded for 65,000 words. Thus, with full expansion, up to 65,000 subroutines can all be going at the same time.

The basic computer is a complete system. The program can be entered via switches located on the front panel, providing a LED readout in binary format. The very-low-cost terminal presented in POPULAR ELECTRONICS last month can also be used.

### PROCESSOR DESCRIPTION

Processor: 8 bit parallel  
Max. memory: 65,000 words (all directly addressable)  
Instruction cycle time: 2  $\mu$ s (min.)  
Inputs and outputs: 256 (all directly addressable)  
Number of basic machine instructions: 78 (181 with variants)  
Add/subtract time: 2  $\mu$ s  
Number of subroutine levels: 65,000  
Interrupt structure: 8 hardwire vectored levels plus software levels  
Number of auxiliary registers: 8 plus stack pointer, program counter and accumulator  
Memory type: semiconductor (dynamic or static RAM, ROM, PROM)  
Memory access time: 850 ns static RAM; 420 or 150 ns dynamic Ram

## 2.7 Formats of Instructions



Lecturer  
Yuanqing  
Cheng

# Computer Architecture (计算机体系结构)

## Lecture 9 MIPS Instruction Representation II

2020-09-21



### Memory Technologies Confront Edge AI's Diverse Challenges

Edge AI applications are many and varied, which means that there are nearly endless options for memory for edge applications.

# Review

---

- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `sw`).
- Computer actually stores programs as a series of these 32-bit numbers.
- **MIPS Machine Language Instruction:** 32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt		immediate	

# I-Format Problems (0/3)

---

- Problem 0: Unsigned # sign-extended?
  - addiu, sltiu, **sign-extends** immediates to 32 bits. Thus, # is a “signed” integer.
- Rationale
  - addiu so that can add w/out overflow
    - See K&R pp. 230, 305
  - sltiu suffers so that we can have easy HW
    - Does this mean we'll get wrong answers?
    - Nope, it means assembler has to handle any unsigned immediate  $2^{15} \leq n < 2^{16}$  (I.e., with a 1 in the 15th bit and 0s in the upper 2 bytes) as it does for numbers that are too large.  $\Rightarrow$

# I-Format Problem (1/3)

---

- Problem:
  - Chances are that addi, lw, sw and slti will use immediates small enough to fit in the immediate field.
  - ...but what if it's too big?
  - We need a way to deal with a 32-bit immediate in any I-format instruction.

# I-Format Problem (2/3)

---

- Solution to Problem:
  - Handle it in software + new instruction
  - Don't change the current instructions: instead, add a new instruction to help out

- New instruction:

lui register, immediate

- stands for Load Upper Immediate
- takes 16-bit immediate and puts these bits in the upper half (high order half) of the register
- sets lower half to 0s

# I-Format Problems (3/3)

---

- Solution to Problem (continued):

- So how does lui help us?
  - Example:

```
addiu $t0,$t0, 0xABABCD
```

...becomes

```
lui $at 0xABAB
ori $at, $at, 0xCD
addu $t0,$t0,$at
```

- Now each I-format instruction has only a 16-bit immediate.
  - Wouldn't it be nice if the assembler would do this for us automatically? (later)

# Branches: PC-Relative Addressing

## (1/5) Use I-Format

opcode	rs	rt	immediate
--------	----	----	-----------

- opcode **specifies** beq **versus** bne
- rs **and** rt **specify registers to compare**
- What can immediate specify?
  - immediate is only 16 bits
  - PC (Program Counter) has byte address of current instruction being executed;  
32-bit pointer to memory
  - So immediate cannot specify entire address to branch to.

# Branches: PC-Relative Addressing

---

(2/5)

How do we typically use branches?

- Answer: if-else, while, for
- Loops are generally small: usually up to 50 instructions
- Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches.
- Conclusion: may want to branch to anywhere in memory, but a branch often changes **PC** by a small amount

# Branches: PC-Relative Addressing

---

(3/5)

■ Solution to branches in a 32-bit instruction:  
**PC-Relative Addressing**

- Let the 16-bit immediate field be a signed two's complement integer to be *added* to the PC if we take the branch.
- Now we can branch  $\pm 2^{15}$  bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

# Branches: PC-Relative Addressing

---

(4/5)

Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary).

- So the number of bytes to add to the PC will always be a multiple of 4.
- So specify the immediate in words.
- Now, we can branch  $\pm 2^{15}$  words from the PC (or  $\pm 2^{17}$  bytes), so we can handle loops 4 times as large.

# Branches: PC-Relative Addressing

---

(5/5)

## Branch Calculation:

- If we **don't** take the branch:

$$PC = PC + 4 = \text{byte address of next instruction}$$

- If we **do** take the branch:

$$PC = (PC + 4) + (\text{immediate} * 4)$$

- Observations

- Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.
- Immediate field can be positive or negative.
- Due to hardware, add immediate to  $(PC+4)$ , not to PC; will be clearer why later in course

# Branch Example (1/3)

---

- MIPS Code:

```
Loop: beq    $9, $0, End
      addu   $8, $8, $10
      addiu  $9, $9, -1
      j       Loop
```

End:

- **beq branch is I-Format:**

opcode = 4 (look up in table)

rs = 9 (first operand)

rt = 0 (second operand)

immediate = ???

# Branch Example (2/3)

---

- MIPS Code:

```
Loop: beq    $9, $0, End
      addu   $8, $8, $10
      addiu  $9, $9, -1
      j      Loop
```

End:

- immediate Field:

- Number of **instructions** to add to (or subtract from) the PC, starting at the instruction *following* the branch.
- In beq **case**, immediate = 3

# Branch Example (3/3)

- MIPS Code:

```
Loop: beq    $9, $0, End
      addu   $8, $8, $10
      addiu  $9, $9, -1
      j       Loop
```

End:

decimal representation:

4	9	0	3
---	---	---	---

binary representation:

000100	01001	00000	0000000000000011
--------	-------	-------	------------------

# Questions on PC-addressing

---

- Does the value in branch field change if we move the code?
- What do we do if destination is  $> 2^{15}$  instructions away from branch?
- Why do we need different addressing modes (different ways of forming a memory address)? Why not just one?

# J-Format Instructions (1/5)

---

- For branches, we assumed that we won't want to branch too far, so we can specify *change* in PC.
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

# J-Format Instructions (2/5)

- Define two “fields” of these bit widths:

6 bits

26 bits

- As usual, each field has a name:

opcode

target address

- Key Concepts

- Keep opcode field identical to R-format and I-format for consistency.
- Collapse all other fields to make room for large target address.

# J-Format Instructions (3/5)

---

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
  - So let's just take this for granted and not even specify them.

# J-Format Instructions (4/5)

---

- Now specify 28 bits of a 32-bit address
- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to *anywhere* in memory, but it's adequate 99.9999...% of the time, since programs aren't that long
    - only if straddle a 256 MB boundary
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the jr instruction.

# J-Format Instructions (5/5)

---

- Summary:
  - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
  - { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 111111111111111111111111, 00 } =  
1010111111111111111111111111111100
  - Note: Book uses ||

# Peer Instruction Question

---

(for A,B) When combining two C files into one executable, recall we can compile them independently & then merge them together.

- 1) **Jump** insts don't require any changes.
- 2) **Branch** insts don't require any changes.

12

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

# In conclusion

---

- MIPS Machine Language Instruction:  
32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt		immediate	
J	opcode		target	address		

- Branches use PC-relative addressing,  
Jumps use absolute addressing.
- Disassembly is simple and starts by  
decoding opcode field. (more in a week)

## 2.8 Float Point

# Computer Architecture (计算机体系结构)

## Lecture 10 Floating Point 1

2020-09-21

Lecturer Yuanqing Cheng  
[www.cadetlab.cn/teaching](http://www.cadetlab.cn/teaching)



## Quote of the day

---

“95% of the  
folks out there are  
**completely clueless**  
about floating-point.”

**James Gosling**  
**Sun Fellow**  
**Java Inventor**  
**1998-02-28**



# Review of Numbers

---

- Computers are made to deal with numbers
- What can we represent in N bits?
  - $2^N$  things, and no more! They could be...
  - Unsigned integers:

0 to  $2^N - 1$

(for N=32,  $2^{32} - 1 = 4,294,967,295$ )

- Signed Integers (Two's Complement)

$-2^{(N-1)}$  to  $2^{(N-1)} - 1$

(for N=32,  $2^{31} = 2,147,483,648$ )

# What about other numbers?

---

1. Very large numbers? (seconds/millennium)  
⇒  $31,556,926,000_{10}$  ( $3.1556926_{10} \times 10^{10}$ )
2. Very small numbers? (Bohr radius)  
⇒  $0.000000000529177_{10}\text{m}$  ( $5.29177_{10} \times 10^{-11}$ )
3. Numbers with both integer & fractional parts?  
⇒ 1.5

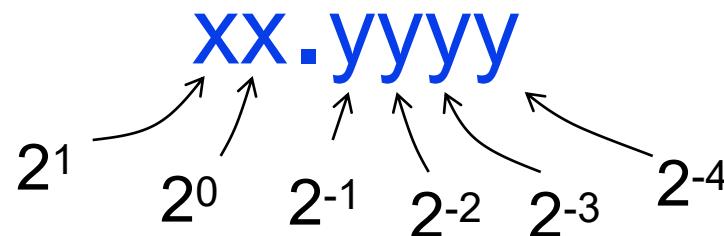
*First consider #3.*

*...our solution will also help with 1 and 2.*

# Representation of Fractions

“Binary Point” like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:



$$10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$$

If we assume “fixed binary point”, range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

# Fractional Powers of 2

---

i	$2^{-i}$	
0	1.0	1
1	0.5	$1/2$
2	0.25	$1/4$
3	0.125	$1/8$
4	0.0625	$1/16$
5	0.03125	$1/32$
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	
11	0.00048828125	
12	0.000244140625	
13	0.0001220703125	
14	0.00006103515625	
15	0.000030517578125	

# Representation of Fractions with Fixed Pt.

What about addition and multiplication?

Addition is  
straightforward:

$$\begin{array}{r} 01.100 \\ + 00.100 \\ \hline 10.000 \end{array} \quad \begin{array}{r} 1.5_{10} \\ 0.5_{10} \\ 2.0_{10} \end{array}$$

Multiplication a bit more complex:

$$\begin{array}{r} 01.100 \\ \times 00.100 \\ \hline 0000110000 \end{array}$$

HI      LOW

Where's the answer, 0.11? (need to remember where point is)

# Representation of Fractions

So far, in our examples we used a “fixed” binary point what we really want is to “float” the binary point. Why?

Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):

example: put 0.1640625 into binary. Represent as in 5-bits choosing where to put the binary point.

... 00000.00101010000...



Store these bits and keep track of the binary point 2 places to the left of the MSB

Any other solution would lose accuracy!

With floating point rep., each numeral carries a exponent field recording the whereabouts of its binary point.

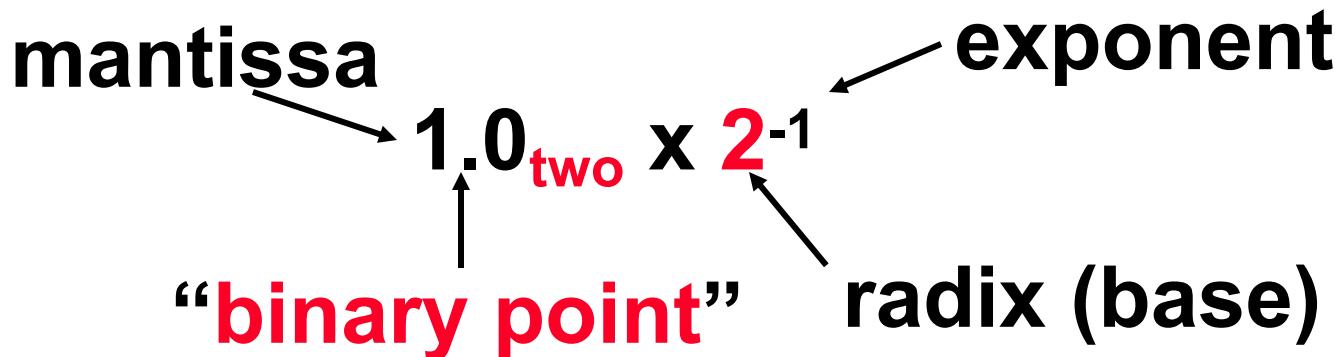
The binary point can be outside the stored bits, so very large and small numbers can be represented.

# Scientific Notation (in Decimal)

**mantissa** →  $6.02_{10} \times 10^{23}$  ← **exponent**  
↑  
**decimal point**      **radix (base)**

- Normalized form: no leadings 0s  
(exactly one digit to left of decimal point)
- Alternatives to representing 1/1,000,000,000
  - Normalized:  $1.0 \times 10^{-9}$
  - Not normalized:  $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

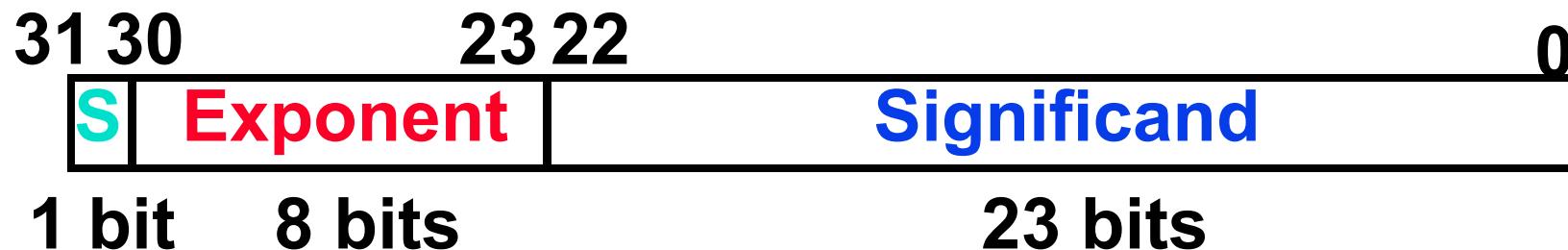
# Scientific Notation (in Binary)



- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`

# Floating Point Representation (1/2)

- Normal format:  $+1.\text{xxx...x}_{\text{two}} * 2^{\text{yyy...y}_{\text{two}}}$
- Multiple of Word Size (32 bits)



- S represents Sign  
Exponent represents y's  
Significand represents x's

- Represent numbers as small as  $2.0 \times 10^{-38}$  to as large as  $2.0 \times 10^{38}$

# Floating Point Representation (2/2)

- What if result too large?

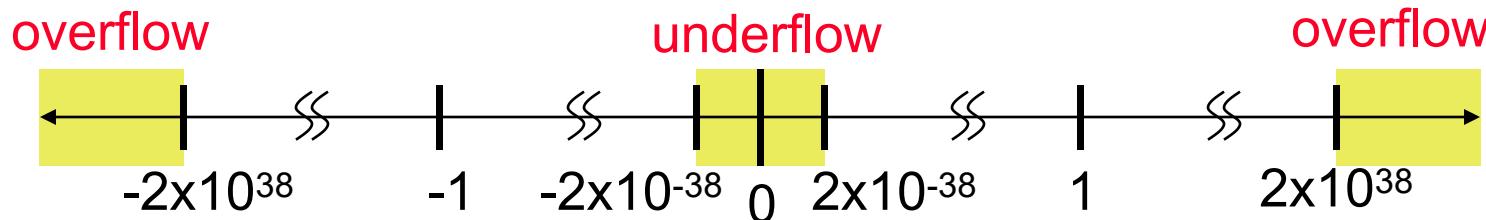
( $> 2.0 \times 10^{38}$ ,  $< -2.0 \times 10^{38}$ )

- **Overflow!**  $\Rightarrow$  Exponent larger than represented in 8-bit Exponent field

- What if result too small?

( $> 0$  &  $< 2.0 \times 10^{-38}$ ,  $< 0$  &  $> -2.0 \times 10^{-38}$ )

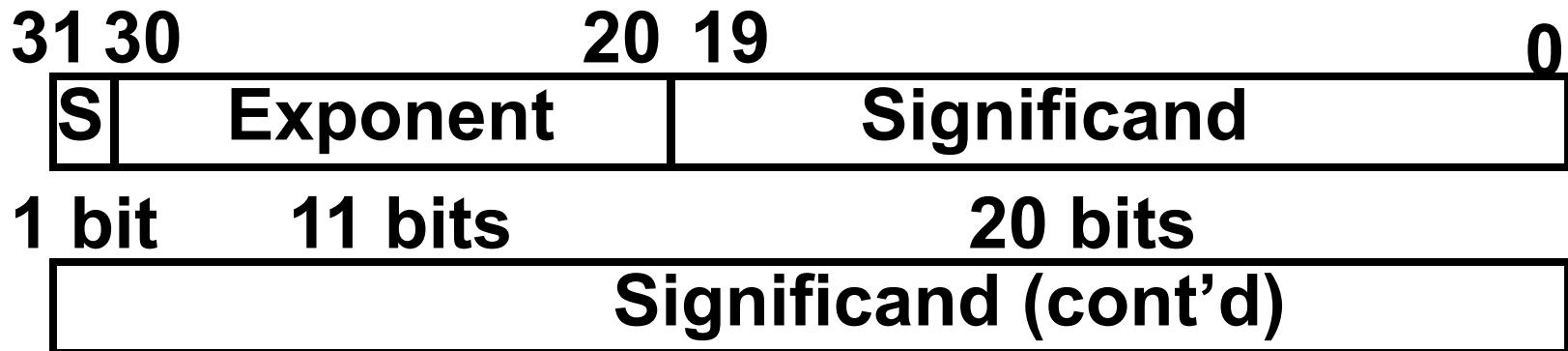
- **Underflow!**  $\Rightarrow$  Negative exponent larger than represented in 8-bit Exponent field



- What would help reduce chances of overflow and/or underflow?

# Double Precision Fl. Pt. Representation

- Next Multiple of Word Size (64 bits)



32 bits

- Double Precision (vs. Single Precision)

- C variable declared as `double`
- Represent numbers almost as small as  $2.0 \times 10^{-308}$  to almost as large as  $2.0 \times 10^{308}$
- But primary advantage is greater accuracy due to larger significand

# QUAD Precision Fl. Pt. Representation

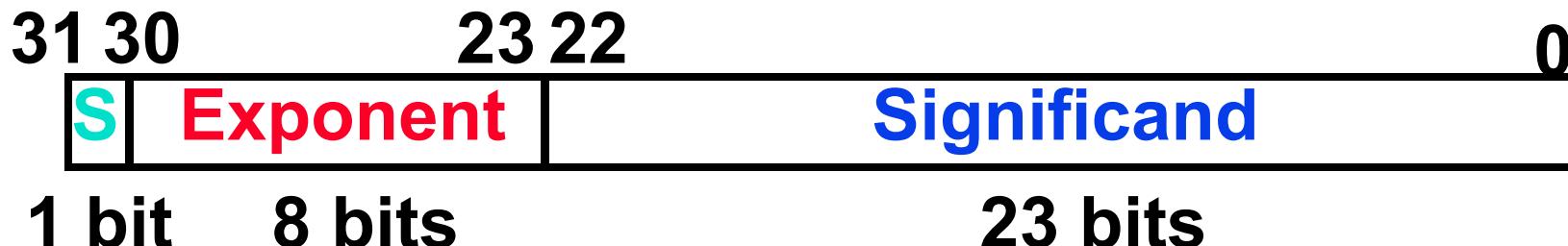
---

- Next Multiple of Word Size (128 bits)
  - Unbelievable range of numbers
  - Unbelievable precision (accuracy)
- IEEE 754-2008 “binary128” standard
  - Has 15 exponent bits and 112 significand bits (113 precision bits)
- Oct-Precision?
  - Some have tried, no real traction so far
- Half-Precision?
  - Yep, “binary16”: 1/5/10

[en.wikipedia.org/wiki/Floating\\_point](https://en.wikipedia.org/wiki/Floating_point)

# IEEE 754 Floating Point Standard (1/3)

Single Precision (DP similar):



- **Sign bit:** 1 means negative  
0 means positive
- **Significand:**
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true:  $0 < \text{Significand} < 1$  (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

# IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation.
  - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
  - Wanted bigger (integer) exponent field to represent bigger numbers.
  - 2's complement poses a problem (because negative numbers look bigger)
  - We're going to see that the numbers are ordered EXACTLY as in sign-magnitude
    - I.e., counting from binary odometer 00...00 up to 11...11 goes from 0 to +MAX to -0 to -MAX to 0

# IEEE 754 Floating Point Standard (3/3)

- Called **Biased Notation**, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

- Summary (single precision):

31	30	23	22	0
S	Exponent		Significand	
1 bit	8 bits		23 bits	

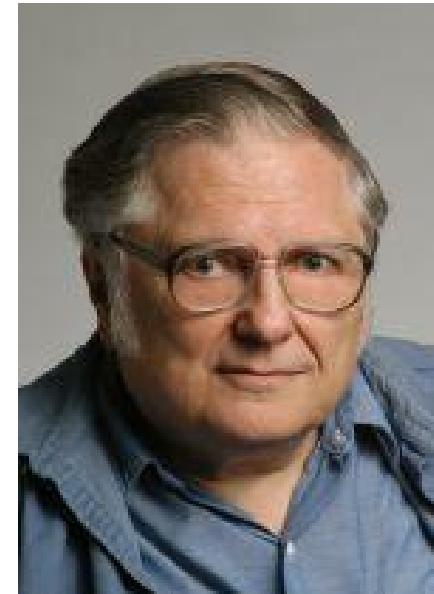
$$\bullet (-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)

# “Father” of the Floating point standard

---

IEEE Standard 754  
for Binary Floating-  
Point Arithmetic.



Prof. Kahan

[www.cs.berkeley.edu/~wkahan/ieee754status/754story.html](http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html)

# Example: Converting Binary FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

- Sign: 0 → positive

- Exponent:

- $0110\ 1000_{\text{two}} = 104_{\text{ten}}$

- Bias adjustment:  $104 - 127 = -23$

- Significand:

$$\begin{aligned} & 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots \\ & = 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} \\ & = 1.0 + 0.666115 \end{aligned}$$

- Represents:  $1.666115_{\text{ten}} \times 2^{-23} \sim 1.986 \times 10^{-7}$   
(about 2/10,000,000)

# Example: Converting Decimal to FP

-2.340625 x 10<sup>1</sup>

1. Denormalize: -23.40625

2. Convert integer part:

$$23 = 16 + ( 7 = 4 + ( 3 = 2 + ( 1 ) ) ) = 10111_2$$

3. Convert fractional part:

$$.40625 = .25 + ( .15625 = .125 + ( .03125 ) ) = .01101_2$$

4. Put parts together and normalize:

$$10111.01101 = 1.011101101 \times 2^4$$

5. Convert exponent:  $127 + 4 = 10000011_2$

1	1000 0011	011 1011 0100 0000 0000 0000
---	-----------	------------------------------

# Peer Instruction

---

**1 1000 0001 | 111 0000 0000 0000 0000 0000**

**What is the decimal equivalent of the floating pt # above?**

- a) -7 \*  $2^{129}$
- b) -3.5
- c) -3.75
- d) -7
- e) -7.5

# Peer Instruction Answer

# What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
S	Exponent	Significand

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

$$(-1)^1 \times (1 + .111) \times 2^{(129-127)}$$

$$-1 \times (1.\textcolor{blue}{111}) \times 2^{\textcolor{red}{(2)}}$$

-111.1

-7.5

- a) -7 \*  $2^{129}$
  - b) -3.5
  - c) -3.75
  - d) -7
  - e) -7.5

# “And in conclusion...”

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

- Summary (single precision):

31	30	23	22	0
S	Exponent		Significand	

1 bit    8 bits                          23 bits

$$\bullet (-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)

# Understanding the Significand (1/2)

---

- Method 1 (Fractions):

- In decimal:  $0.340_{10}$        $\Rightarrow 340_{10}/1000_{10}$   
 $\Rightarrow 34_{10}/100_{10}$

- In binary:  $0.110_2 \Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$   
 $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$

- Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

# Understanding the Significand (2/2)

---

- Method 2 (Place Values):
  - Convert from scientific notation
  - In decimal:  $1.6732 = (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
  - In binary:  $1.1001 = (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
  - Interpretation of value in each position extends beyond the decimal/binary point
  - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

# Computer Architecture (计算机体系结构)

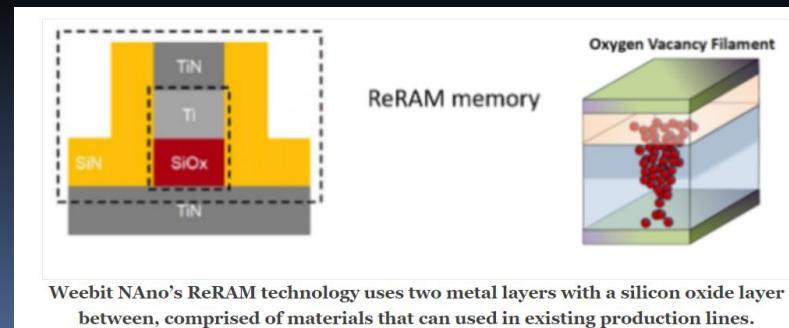
## Lecture 11 Floating Point II

2020-09-25



Lecturer Yuanqing Cheng  
[www.cadetlab.cn/teaching](http://www.cadetlab.cn/teaching)

Emerging Memories May Never Go  
Beyond Niche Applications



# Review

Exponent tells Significand how much ( $2^i$ ) to count by (... ,  $1/4$ ,  $1/2$ ,  $1$ ,  $2$ , ...)

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

Summary (single precision):



$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

Double precision identical, except with exponent bias of 1023 (half, quad similar)

# “Father” of the Floating point standard

---

IEEE Standard  
754 for Binary  
Floating-Point  
Arithmetic.



1989 ACM Turing  
Award Winner!

Prof. Kahan

[www.cs.berkeley.edu/~wkahan/  
.../ieee754status/754story.html](http://www.cs.berkeley.edu/~wkahan/.../ieee754status/754story.html)

# Precision and Accuracy

*Don't confuse these two terms!*

---

**Precision** is a count of the number bits in a computer word used to represent a value.

**Accuracy** is a measure of the difference between the actual value of a number and its computer representation.

*High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.*

*Example:*    `float pi = 3.14;`

pi will be represented using all 24 bits of the significant (highly precise), but is only an approximation (not accurate).

# Representation for $\pm \infty$

---

- In FP, divide by 0 should produce  $\pm \infty$ , not overflow.
- Why?
  - OK to do further computations with  $\infty$  E.g.,  $X/0 > Y$  may be a valid comparison
  - Ask math majors
- IEEE 754 represents  $\pm \infty$ 
  - Most positive exponent reserved for  $\infty$
  - Significands all zeroes

# Representation for 0

---

- Represent 0?
    - **exponent all zeroes**
    - **significand all zeroes**
    - **What about sign? Both cases valid.**
- +0: 0 00000000 00000000000000000000000000000000
- 0: 1 00000000 00000000000000000000000000000000

# Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	???
1-254	anything	+/- fl. pt. #
255	0	+/- $\infty$
255	<u>nonzero</u>	???

- Professor Kahan had clever ideas;  
“Waste not, want not”
  - We'll talk about  $\text{Exp}=0, 255$  &  $\text{Sig}!=0$  later

# Representation for Not a Number

---

- What do I get if I calculate  
 $\sqrt{-4.0}$  or  $0/0$ ?
  - If  $\infty$  not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate:  $op(\text{NaN}, X) = \text{NaN}$

# Representation for Denorms (1 / 2)

- Problem: There's a gap among representable FP numbers around 0

- Smallest representable pos num:

$$a = 1.0\ldots_2 \cdot 2^{-126} = 2^{-126}$$

- Second smallest representable pos num:

$$b = 1.000\ldots1_2 \cdot 2^{-126}$$

$$= (1 + 0.00\ldots1_2) \cdot 2^{-126}$$

$$= (1 + 2^{-23}) \cdot 2^{-126}$$

$$= 2^{-126} + 2^{-149}$$

$$a - 0 = 2^{-126}$$

$$b - a = 2^{-149}$$



Normalization and  
implicit 1  
is to blame!

# Representation for Denorm (2/2)

---

- Solution:
  - We still haven't used Exponent=0, Significand nonzero
  - Denormalized number: no (implied) leading 1, implicit exponent = -126
  - Smallest representable pos num:
    - $A = 2^{-149}$
  - Second smallest representable pos num:
    - $b = 2^{-148}$



# Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>Denorm</u>
1-254	Anything	+/- fl. Pt #
255	<u>0</u>	<u>+/- <math>\infty</math></u>
255	<u>nonzero</u>	<u>NaN</u>

# Rounding

---

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:  
**double to a single precision value, or floating point number to an integer**

# IEEE FP Rounding Modes

---

- Halfway between two floating point values (rounding bits read 10)? Choose from the following:
  - Round towards  $+\infty$ 
    - Round “up”:  $1.01\ \underline{10} \rightarrow 1.10$  ,  $-1.01\ \underline{10} \rightarrow -1.01$
  - Round towards  $-\infty$ 
    - Round “down”:  $1.01\ \underline{10} \rightarrow 1.01$ ,  $-1.01\ \underline{10} \rightarrow -1.10$
  - Truncate
    - Just drop the extra bits (round towards 0)
- Unbiased (default mode). Round to nearest EVEN number
  - Half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies.
  - In binary, even means least significant bit is 0.
- Otherwise, not halfway (00, 01, 11)! Just round to the nearest float.

# Peer Instruction

---

1. Converting float -> int -> float produces same float number
2. Converting int -> float -> int produces same int number
3. FP add is associative:  
 $(x+y)+z = x+(y+z)$

ABC

1 : FFF

2 : FFT

3 : FTF

4 : FTT

5 : TFF

# Peer Instruction Answer

1. Converting float > int > float produces same float number

**FALSE**

2. Converting int -> float > int produces same int number

**FALSE**

3. FP addition associative.

$(x+y) + z = x + (y+z)$

**FALSE**

1.  $3.14 \rightarrow 3 \rightarrow 3$

2. 32 bits for signed int,  
but 24 for FP mantissa?

3.  $x = \text{biggest pos \#},$   
 $y = -x, z = 1 (x \neq \text{inf})$

ABC

1: **FFF**

2: **FFT**

3: **FTF**

4: **FTT**

5: **TFF**

# Peer Instruction

---

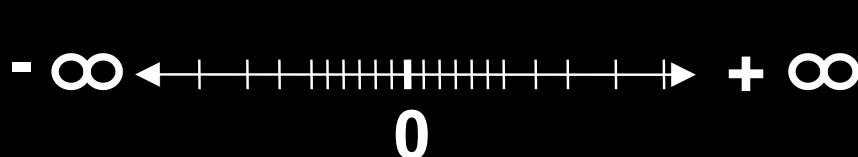
- Let  $f(1, 2) = \# \text{ of floats between } 1 \text{ and } 2$
- Let  $f(2, 3) = \# \text{ of floats between } 2 \text{ and } 3$

1:  $f(1, 2) < f(2, 3)$   
2:  $f(1, 2) = f(2, 3)$   
3:  $f(1, 2) > f(2, 3)$

# Peer Instruction Answer

---

- Let  $f(1, 2) = \# \text{ of floats between } 1 \text{ and } 2$
- Let  $f(2, 3) = \# \text{ of floats between } 2 \text{ and } 3$



1:  $f(1, 2) < f(2, 3)$   
2:  $f(1, 2) = f(2, 3)$   
3:  $f(1, 2) > f(2, 3)$

# “And in conclusion...”

---

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>Denorm</u>
1-254	Anything	+/- fl. Pt #
255	0	+/- $\infty$
255	<u>nonzero</u>	<u>NaN</u>

- 4 Rounding modes (default: unbiased)
- MIPS Fl ops complicated, expensive

# Bonus slides

---

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus

# FP Addition

---

- More difficult than with integers
- Can't just add significands
- How do we do it?
  - De-normalize to match exponents
  - Add significands to get resulting one
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- Note: If signs differ, just perform a subtract instead.

# MIPS Floating Point Architecture

(1/4)

---

- MIPS has special instructions for floating point operations:
  - **Single Precision:**  
add.s, sub.s, mul.s, div.s
  - **Double Precision:**  
add.d, sub.d, mul.d, div.d
- These instructions are far more complicated than their integer counterparts. They require special hardware and usually they can take much longer to compute.

# MIPS Floating Point Architecture

(2 / 4)

---

## • Problems:

- It's inefficient to have different instructions take vastly differing amounts of time.
- Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.
- Some programs do no floating point calculations
- It takes lots of hardware relative to integers to do Floating Point fast

# MIPS Floating Point Architecture

(3/4)

---

- 1990 Solution: Make a completely separate chip that handles only FP.
  - **Coprocessor 1: FP chip**
    - **contains 32 32-bit registers:** \$f0, \$f1, ...
    - **most registers specified in .s and .d instruction refer to this set**
    - **separate load and store:** lwcl and swcl (“load word coprocessor 1”, “store ...”)
    - **Double Precision: by convention, even/odd pair contain one DP FP number:** \$f0/\$f1, \$f2/\$f3, ... , \$f30/\$f31

# MIPS Floating Point Architecture

(4/4) Computer actually contains multiple separate chips:

- Processor: handles all the normal stuff
- Coprocessor 1: handles FP and only FP;
- more coprocessors?... Yes, later
- Today, cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - mfc0, mtc0, mfc1, mtc1, etc.
- Appendix pages A-70 to A-74 contain many, many more FP operations.

# Example: Representing 1/3 in

MIPS  
1/3

$$= 0.33333\dots_{10}$$

$$= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \dots$$

$$= 1/4 + 1/16 + 1/64 + 1/256 + \dots$$

$$= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \dots$$

$$= 0.0101010101\dots_2 * 2^0$$

$$= 1.0101010101\dots_2 * 2^{-2}$$

- **Sign:** 0
- **Exponent = -2 + 127 = 125 = 01111101**
- **Significand = 0101010101...**

0	0111 1101	0101 0101 0101 0101 0101 010
---	-----------	------------------------------

# Casting floats to ints and vice

versa

**(int) *floating\_point\_expression***

**Coerces and converts it to the nearest integer (C uses truncation)**

```
i = (int) (3.14159 * f);
```

**(float) *integer\_expression***

**converts integer to nearest floating point**

```
f = f + (float) i;
```

int → float → int

---

```
if (i == (int)((float) i)) {  
    printf("true");  
}
```

- Will not always print “true”
- Most large values of integers don’t have exact floating point representations!
- What about double?

# float → int → float

---

```
if (f == (float)((int) f)) {  
    printf("true");  
}
```

- Will not always print “true”
- Small floating point numbers (<1) don’t have integer representations
- For other numbers, rounding errors

# Floating Point Fallacy

---

- FP add associative: FALSE!
  - $x = -1.5 \times 10^{38}$ ,  $y = 1.5 \times 10^{38}$ , and  $z = 1.0$
  - $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$   
 $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$
  - $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$   
 $= (0.0) + 1.0 = \underline{1.0}$
- Therefore, Floating Point add is not associative!
  - Why? FP result approximates real result!
  - This example:  $1.5 \times 10^{38}$  is so much larger than  $1.0$  that  $1.5 \times 10^{38} + 1.0$  in floating point representation is still  $1.5 \times 10^{38}$

## 2.9 Instruction Represation

# Computer Architecture (计算机体系结构)

## Lecture 12 Instruction Representation III



2020-09-25

Lecturer Yuanqing Cheng

[www.cadetlab.cn/~courses](http://www.cadetlab.cn/~courses)

# Review

---

- **MIPS Machine Language Instruction:**  
32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt		immediate	
J	opcode		target	address		

- Branches use PC-relative addressing,  
Jumps use absolute addressing.

# Outline

---

- Disassembly
- Pseudoinstructions
- “True” Assembly Language (TAL) vs.  
“MIPS” Assembly Language (MAL)

# Decoding Machine Language

---

- How do we convert 1s and 0s to assembly language and to C code?  
Machine language  $\Rightarrow$  assembly  $\Rightarrow$  C?
- For each 32 bits:
  1. Look at opcode to distinguish between R-Format, J-Format, and I-Format.
  2. Use instruction format to determine which fields exist.
  3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
  4. Logically convert this MIPS code into valid C code. Always possible? Unique?

# Decoding Example (1/7)

---

- Here are six machine language instructions in hexadecimal:

00001025<sub>hex</sub>

0005402A<sub>hex</sub>

11000003<sub>hex</sub>

00441020<sub>hex</sub>

20A5FFFF<sub>hex</sub>

08100001<sub>hex</sub>

- Let the first instruction be at address 4,194,304<sub>ten</sub> (0x00400000<sub>hex</sub>).
- Next step: convert hex to binary

# Decoding Example (2/7)

- The six machine language instructions in binary:

```
0000000000000000000000001000000100101  
00000000000000001010100000000101010  
000100010000000000000000000000000011  
000000000010001000001000000100000  
00100000101001011111111111111111  
0000100000010000000000000000000001
```

- Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4-62	rs	rt		immediate	
J	2 or 3		target	address		

# Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:

R	000000	000000	000000	00010	00000	100101
R	000000	000000	000101	01000	00000	101010
I	000100	01000	000000	000000000000000011		
R	000000	000100	0100000010	000000	100000	
I	001000	00101	00101111111111111111			
J	000010	000001000000000000000001				

- Look at opcode:  
0 means R-Format,  
2 or 3 mean J-Format,  
otherwise I-Format.
- Next step: separation of fields

# Decoding Example (4/7)

- Fields separated based on format/opcode:

Format:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0		+3	
R	0	2	4	2	0	32
I	8	5	5		-1	
J	2			1,048,577		

- Next step: translate (“disassemble”) to MIPS assembly instructions

# Decoding Example (5/7)

---

- MIPS Assembly (Part 1):

Address:	Assembly instructions:
----------	------------------------

0x00400000	or	\$2, \$0, \$0
0x00400004	slt	\$8, \$0, \$5
0x00400008	beq	\$8, \$0, 3
0x0040000c	add	\$2, \$2, \$4
0x00400010	addi	\$5, \$5, -1
0x00400014	j	0x100001

- Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

# Decoding Example (6/7)

---

- MIPS Assembly (Part 2):

	or	\$v0, \$0, \$0
Loop:	slt	\$t0, \$0, \$a1
	beq	\$t0, \$0, Exit
	add	\$v0, \$v0, \$a0
	addi	\$a1, \$a1, -1
	j	Loop

Exit:

- Next step: translate to C code  
(must be creative!)

# Decoding Example (7/7)

Before Hex: • After C code (Mapping below)

00001025<sub>hex</sub>  
0005402A<sub>hex</sub>  
11000003<sub>hex</sub>  
00441020<sub>hex</sub>  
20A5FFFF<sub>hex</sub>  
08100001<sub>hex</sub>

\$v0: product  
\$a0: multiplicand  
\$a1: multiplier

```
product = 0;  
while (multiplier > 0) {  
    product += multiplicand;  
    multiplier -= 1;  
}
```

or \$v0,\$0,\$0  
Loop: slt \$t0,\$0,\$a1  
 beq \$t0,\$0,Exit  
 add \$v0,\$v0,\$a0  
 addi \$a1,\$a1,-1  
 j Loop  
  
Exit:

Demonstrated Big Idea: Instructions are just numbers, code is treated like data

# Review from before: lui

---

- So how does lui help us?

- Example:

addi \$t0,\$t0, 0xABABCD

becomes:

lui	\$at, 0xABAB
ori	\$at, \$at, 0xCD
add	\$t0,\$t0,\$at

- Now each I-format instruction has only a 16-bit immediate.
  - Wouldn't it be nice if the assembler would do this for us automatically?
    - If number too big, then just automatically replace addi with lui, ori, add

# True Assembly Language (1/3)

---

- **Pseudoinstruction:** A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions
- What happens with pseudo-instructions?
  - They're broken up by the assembler into several “real” MIPS instructions.
  - Some examples follow

# Example Pseudoinstructions

---

- **Register Move**

**move reg2,reg1**

**Expands to:**

**add reg2,\$zero,reg1**

- **Load Immediate**

**li reg,value**

**If value fits in 16 bits:**

**addi reg,\$zero,value**

**else:**

**lui reg,upper 16 bits of value**

**ori reg,reg,lower 16 bits**

# Example Pseudoinstructions

---

- **Load Address:** How do we get the address of an instruction or global variable into a register?

`la reg, label`

Again if value fits in 16 bits:

`addi reg, $zero, label_value`

**else:**

`lui reg, upper 16 bits of value`

`ori reg, reg, lower 16 bits`

# True Assembly Language (2/3)

---

- **Problem:**
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses any regular register, it'll overwrite whatever the program has put into it.
- **Solution:**
  - Reserve a register (\$1, called \$at for “assembler temporary”) that assembler will use to break up pseudo-instructions.
  - Since the assembler may use this at any time, it's not safe to code with it.

# Example Pseudoinstructions

- Rotate Right Instruction

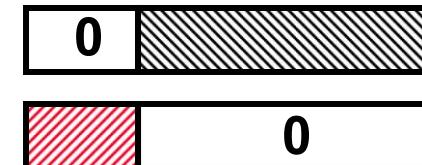
**ror reg, value**

Expands to:

**srl \$at, reg, value**



**sll reg, reg, 32-value**



**or reg, reg, \$at**



- “No OPeration” instruction

**nop**

Expands to instruction = 0<sub>ten</sub>,

**sll \$0, \$0, 0**

# Example Pseudoinstructions

---

- Wrong operation for operand

`addu reg,reg,value # should be addiu`

If value fits in 16 bits, addu is changed to:

`addiu reg,reg,value`

`else:`

`lui $at,upper 16 bits of value`

`ori $at,$at,lower 16 bits`

`addu reg,reg,$at`

- How do we avoid confusion about whether we are talking about MIPS assembler with or without pseudoinstructions?

# True Assembly Language (3/3)

---

- **MAL** (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes pseudoinstructions
- **TAL** (True Assembly Language): set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)
- A program must be converted from MAL into TAL before translation into 1s & 0s.

# Questions on Pseudoinstructions

---

- **Question:**
  - How does MIPS assembler / SPIM recognize pseudo-instructions?
- **Answer:**
  - It looks for officially defined pseudo-instructions, such as `nor` and `move`
  - It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully

# Rewrite TAL as MAL

---

- TAL:

	or	\$v0,\$0,\$0
Loop:	slt	\$t0,\$0,\$a1
	beq	\$t0,\$0,Exit
	add	\$v0,\$v0,\$a0
	addi	\$a1,\$a1,-1
	j	Loop

Exit:

- This time convert to MAL
- It's OK for this exercise to make up MAL instructions

# Rewrite TAL as MAL (Answer)

---

- **TAL:**

Loop:

or	\$v0,\$0,\$0
slt	\$t0,\$0,\$a1
beq	\$t0,\$0,Exit
add	\$v0,\$v0,\$a0
addi	\$a1,\$a1,-1
j	Loop

Exit:

- **MAL:**

Loop:

li	\$v0,0
ble	\$a1,\$zero,Exit
add	\$v0,\$v0,\$a0
sub	\$a1,\$a1,1
j	Loop

Exit:

# Peer Instruction

---

- Which of the instructions below are **MAL** and which are **TAL**?

1. addi \$t0, \$t1, 40000
2. beq \$s0, 10, Exit

	12
a)	MM
b)	MT
c)	TM
d)	TT

# Peer Instruction Answer

---

- Which of the instructions below are **MAL** and which are **TAL**?

1. addi \$t0, \$t1, **40000**  $40,000 > +32,767 \Rightarrow \text{lui,ori}$
2. beq \$s0, **10, Exit** Beq: both must be registers  
Exit: if  $> 2^{15}$ , then MAL

12
a) <b>MM</b>
b) <b>MT</b>
c) <b>TM</b>
d) TT

# In Conclusion

---

- Disassembly is simple and starts by decoding opcode field.
  - Be creative, efficient when authoring C
- Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)
  - Only TAL can be converted to raw binary
  - Assembler's job to do conversion
  - Assembler uses reserved register \$at
  - MAL makes it much easier to write MIPS

## 2.10 Running a Program

# Computer Architecture (计算机体系结构)

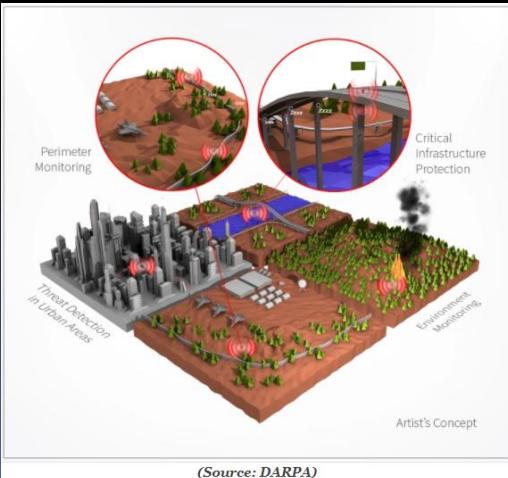


Lecturer  
Yuanqing

## Lecture 13 – Running a Program I (Compiling, Assembling, Linking, Loading)

2020.09.28

### DARPA: Research Advances for Near-Zero-Power Sensors



# Review

---

- Disassembly is simple and starts by decoding opcode field.
  - Be creative, efficient when authoring C
- Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)
  - Only TAL can be converted to raw binary
  - Assembler's job to do conversion
  - Assembler uses reserved register \$at
  - MAL makes it much easier to write MIPS

# Overview

---

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker (next time)
  - Loader (next time)
- An Example (next time)

# Language Execution Continuum

- An Interpreter is a program that executes other programs.



- Language **translation** gives us another option.
- In general, we **interpret** a high level language when efficiency is not critical and **translate** to a lower level language

# Interpretation vs Translation

---

- How do we run a program written in a source language?
  - **Interpreter**: Directly executes a program in the source language
  - **Translator**: Converts a program from the source language to an equivalent program in another language
- For example, consider a Scheme program `foo.scm`

# Interpretation

---

Scheme program: `foo.scm`

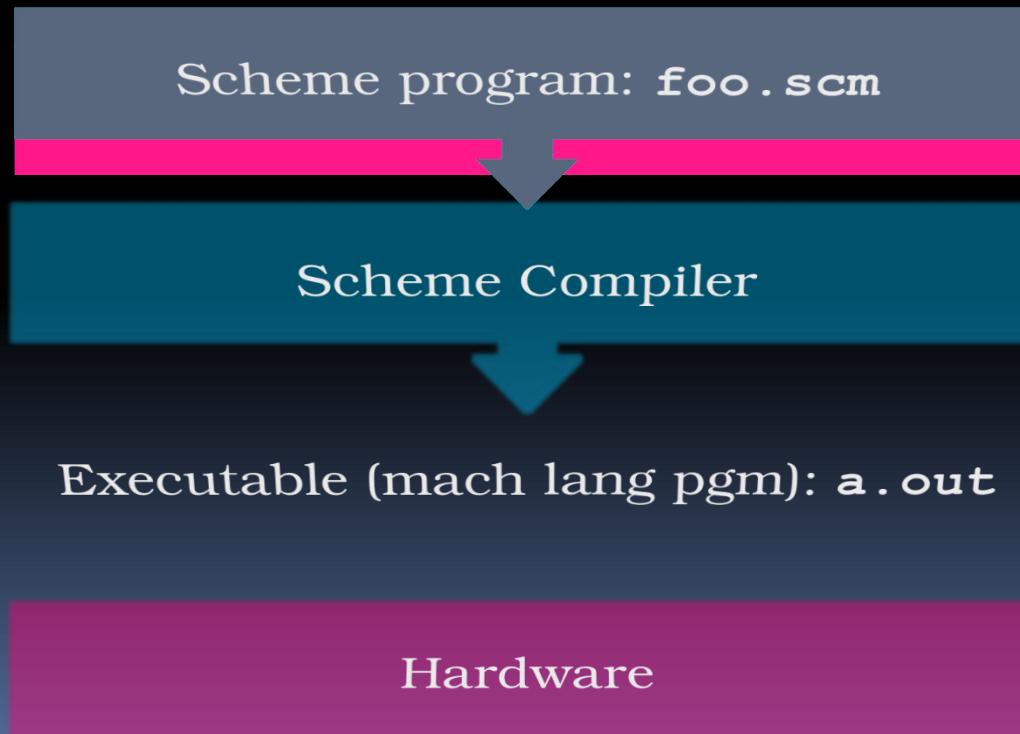
Scheme interpreter

- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

# Translation

---

- Scheme Compiler is a translator from Scheme to machine language.
- The processor is a hardware interpreter of machine language.



# Interpretation

---

- Any good reason to interpret machine language in software?
- SPIM – useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switch to x86.
    - Could require all programs to be re-translated from high level language
    - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)

# Interpretation vs. Translation?

---

- Generally easier to write interpreter
  - Interpreter closer to high-level, so can give better error messages (e.g., MARS, stk)
    - Translator reaction: add extra information to help debugging (line numbers, names)
  - Interpreter slower (10x?), code smaller (2x?)
  - Interpreter provides instruction set independence: run on any machine

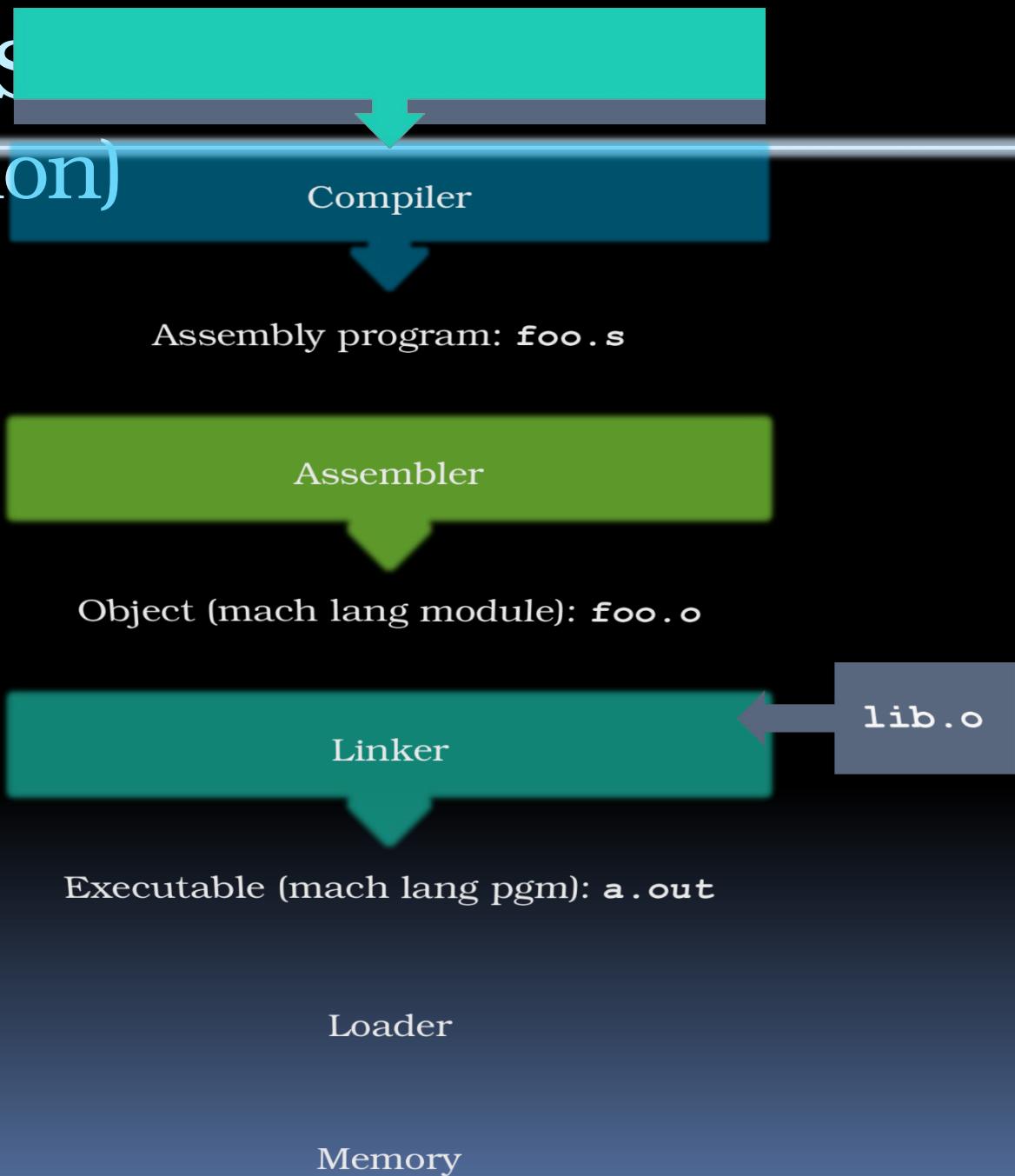
# Interpretation vs. Translation?

---

Translated/compiled code almost always more efficient and therefore higher performance:

- Important for many applications, particularly operating systems.
- Translation/compilation helps “hide” the program “source” from the users:
  - One model for creating value in the marketplace (eg. Microsoft keeps all their source code secret)
  - Alternative model, “open source”, creates value by publishing the source code and fostering a community of developers.

# Steps to S (translation)



# Compiler

---

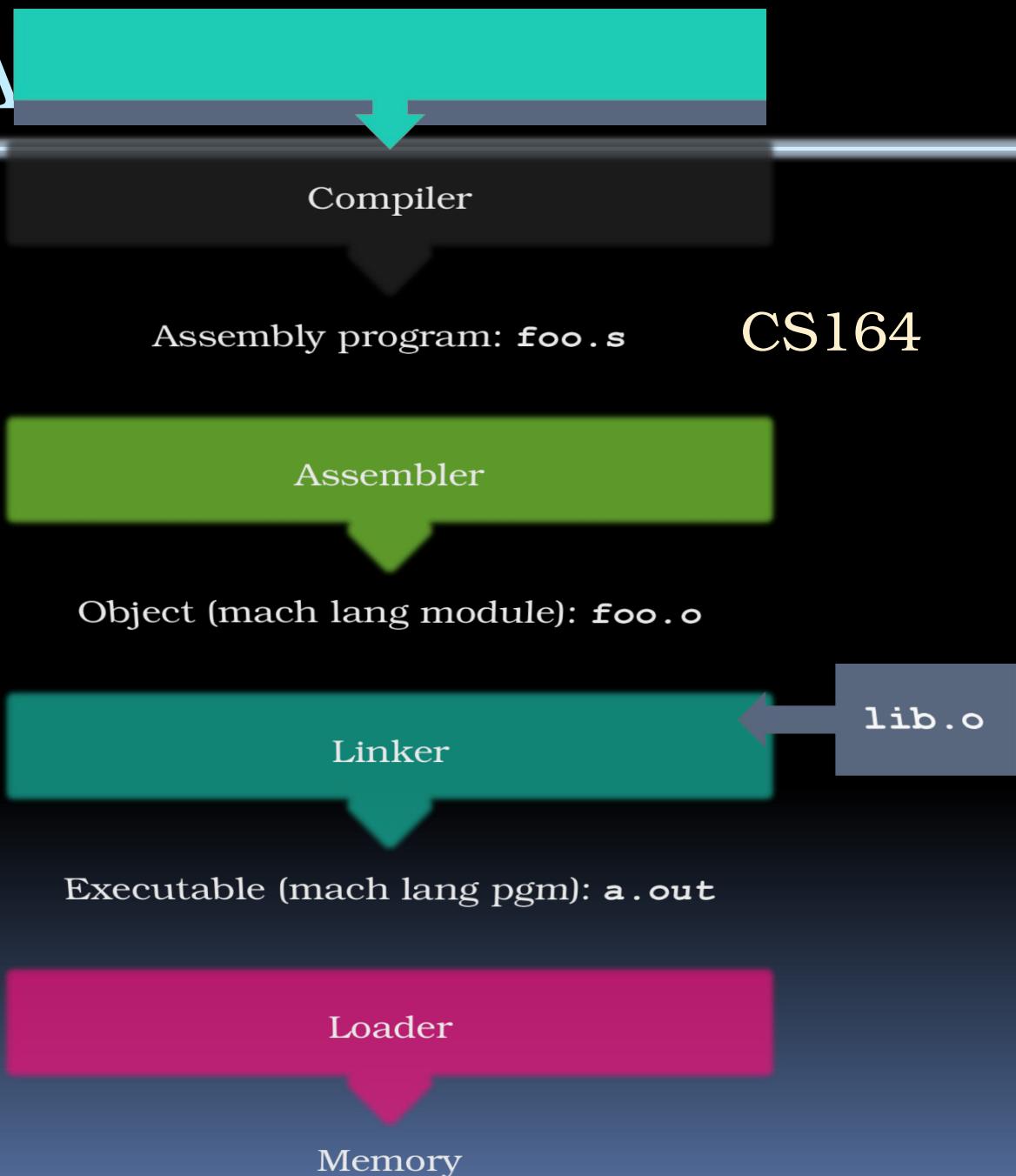
- Input: High-Level Language Code (e.g., C, Java such as `foo.c`)
- Output: Assembly Language Code (e.g., `foo.s` for MIPS)
- Note: Output *may* contain pseudoinstructions
- Pseudoinstructions: instructions that assembler understands but not in machine (last lecture)  
For example:
  - `mov $s1,$s2` ⇒ `or $s1,$s2,$zero`

# Administrivia...

---

- Deadline of assignment: 10.9
- Deadline of lab 1 & lab 2: 10.9

# Where Are We?



# Assembler

---

- Input: Assembly Language Code  
(e.g., `foo.s` for MIPS)
- Output: Object Code, information tables  
(e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

# Assembler Directives (p. A-51 to

---

## A-53)

■ Give directions to assembler, but do not produce machine instructions

.text: Subsequent items put in user text segment (machine code)

.data: Subsequent items put in user data segment (binary rep of data in source file)

.globl sym: declares sym global and can be referenced from other files

.asciiz str: Store the string str in memory and null-terminate it

.word w1...wn: Store the  $n$  32-bit quantities in successive memory words

# Pseudoinstruction Replacement



**subu \$sp,\$sp,32**

sd \$a0, 32(\$sp)

**mul \$t7,\$t6,\$t5**

**addu \$t0,\$t6,1**

ble \$t0,100,loop

la \$a0, str

Real:

**addiu \$sp,\$sp,-32**

**sw \$a0, 32(\$sp)**

**sw \$a1, 36(\$sp)**

mul \$t6,\$t5

**mflo \$t7**

addiu \$t0,\$t6,1

**slti \$at,\$t0,101**

bne \$at,\$0,loop

**lui \$at, left(str)**

ori \$a0,\$at,right(str)

# Producing Machine Language

---

## ■ 1 Simple Case

- Arithmetic, Logical, Shifts, and so on.
- All necessary info is within the instruction already.
- What about Branches?
  - PC-Relative
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch.
- So these can be handled.

# Producing Machine Language

---

## (2“/3)forward Reference” problem

- Branch instructions can refer to labels that are “forward” in the program:

```
        or    $v0, $0, $0
L1: slt   $t0, $0, $a1
     beq   $t0, $0, L2
     addi  $a1, $a1, -1
     j     L1
L2: add   $t1, $a0, $a1
```

- Solved by taking 2 passes over the program.
  - First pass remembers position of labels
  - Second pass uses label positions to generate code

# Producing Machine Language

---

(3/3)

## ■ What about jumps (`j` and `jal`)?

- Jumps require **absolute address**.
- So, forward or not, still can't generate machine instruction without knowing the position of instructions in memory.
- What about references to data?
  - `la` gets broken up into `lui` and `ori`
  - These will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables...

# Symbol Table

---

- List of “items” in this file that may be used by other files.
- What are they?
  - Labels: function calling
  - Data: anything in the **.data** section; variables which may be accessed across files

# Relocation Table

---

- List of “items” this file needs the address later.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any piece of data
    - such as the `la` instruction

# Object File Format

---

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be “handled”
- symbol table: list of this file’s labels and data that can be referenced
- debugging information
- A standard format is ELF (except MS)

[http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)

# Peer Instruction

---

- 1) Assembler will **ignore** the instruction **Loop:nop** because it does nothing.
- 2) Java designers used a translater AND interpreter (rather than just a translater) **mainly** because of (at least 1 of): ease of writing, better error msgs, smaller object code.

	12
a)	FF
b)	FT
c)	TF
d)	TT

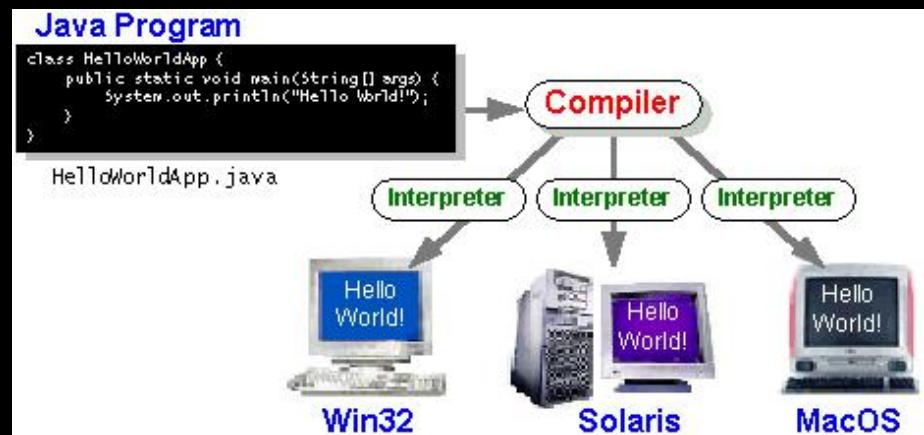
# Peer Instruction Answer

1) Assembler keeps track of all labels in symbol table...F!

2) Java designers used both mainly because of code portability...F!

1) Assembler will **ignore** the instruction **Loop:nop** because it does nothing.

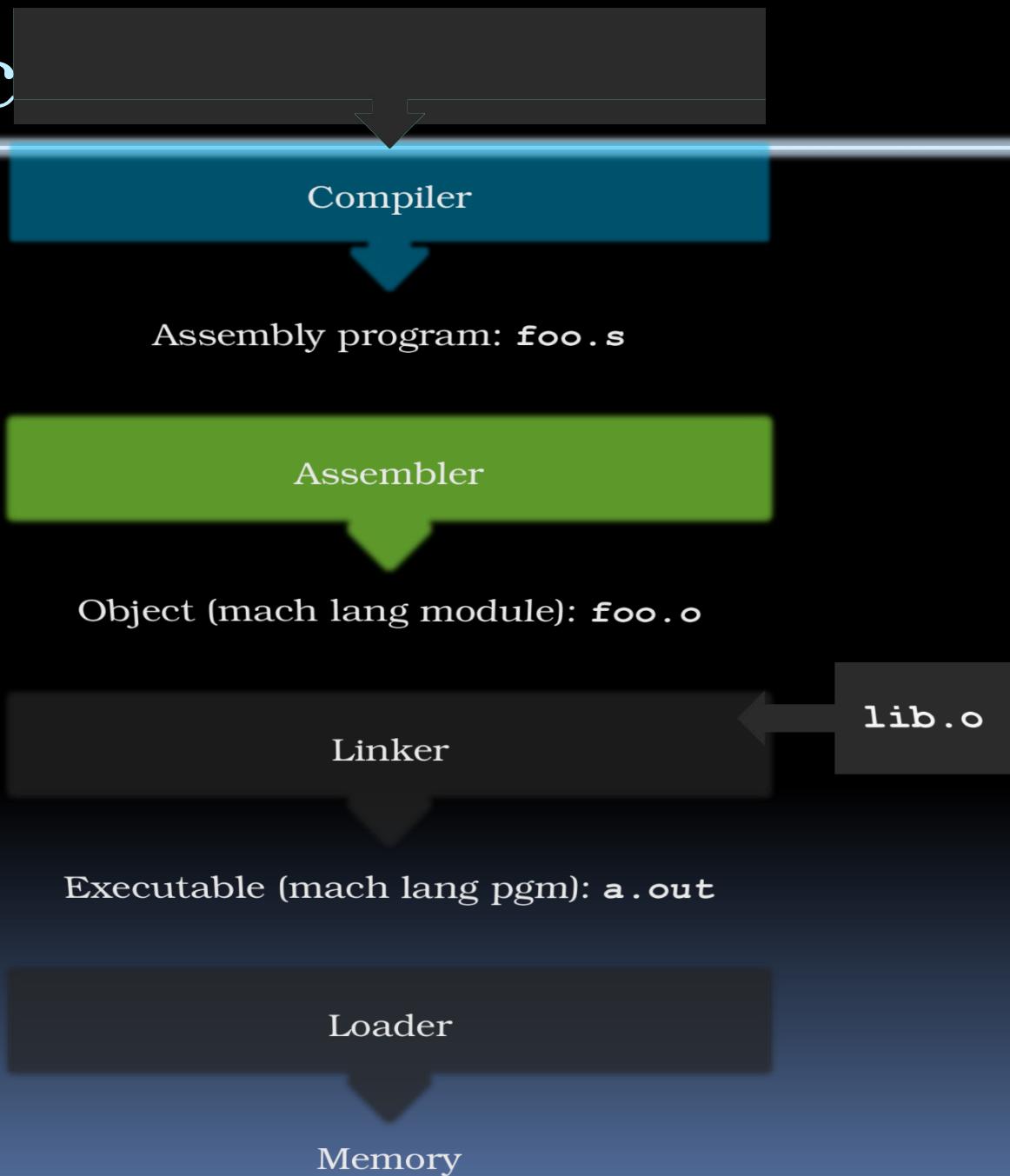
2) Java designers used a translater AND interpreter (rather than just a translater) **mainly** because of (at least 1 of): ease of writing, better error msgs, smaller object code.



12

- a) FF
- b) FT
- c) TF
- d) TT

# And in c



# Bonus slides

---

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus

# Integer Multiplication

---

- (1/3) Paper and pencil example (unsigned):

Multiplicand	1000	8
Multiplier	$\begin{array}{r} \times 1001 \\ \hline \end{array}$	9
	1000	
	0000	
	0000	
	$+1000$	
	$\hline$	
	01001000	

- $m \text{ bits} \times n \text{ bits} = m + n \text{ bit product}$

# Integer Multiplication (2 / 3)

---

- In MIPS, we multiply registers, so:
  - 32-bit value  $\times$  32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult register1, register2`
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product **upper half in hi, lower half in lo**
  - **hi** and **lo** are 2 registers separate from the 32 general purpose registers
  - Use **mfhi** register & **mflo** register to move **from hi, lo** to another register

# Integer Multiplication (3/3)

---

- Example:
  - in C:  $a = b * c;$
  - in MIPS:
    - let  $b$  be  $\$s2$ ; let  $c$  be  $\$s3$ ; and let  $a$  be  $\$s0$  and  $\$s1$  (since it may be up to 64 bits)

```
mult $s2,$s3      # b*c
mfhi $s0          # upper half of
                  # product into $s0
mflo $s1          # lower half of
                  # product into $s1
```

- Note: Often, we only care about the lower half of the product.

# Integer Division (1 / 2)

---

- Paper and pencil example (unsigned):

$$\begin{array}{r} & \underline{1001} & \text{Quotient} \\ \text{Divisor } 1000 | & 1001010 & \text{Dividend} \\ & \underline{-1000} \\ & 10 \\ & 101 \\ & 1010 \\ & \underline{-1000} \\ & 10 & \text{Remainder} \\ & & (\text{or Modulo result}) \end{array}$$

- $\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$

# Integer Division (2/2)

---

- Syntax of Division (signed):
  - `div register1, register2`
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (`/`) and modulo (`%`)
- Example in C:
$$a = c / d; \quad b = c \% d;$$
- in MIPS:
$$a \leftrightarrow \$s0; b \leftrightarrow \$s1; c \leftrightarrow \$s2; d \leftrightarrow \$s3$$

```
div $s2, $s3      # lo=c/d, hi=c%d  
mflo $s0          # get quotient
```



# Computer Architecture

(计算机体系结构)

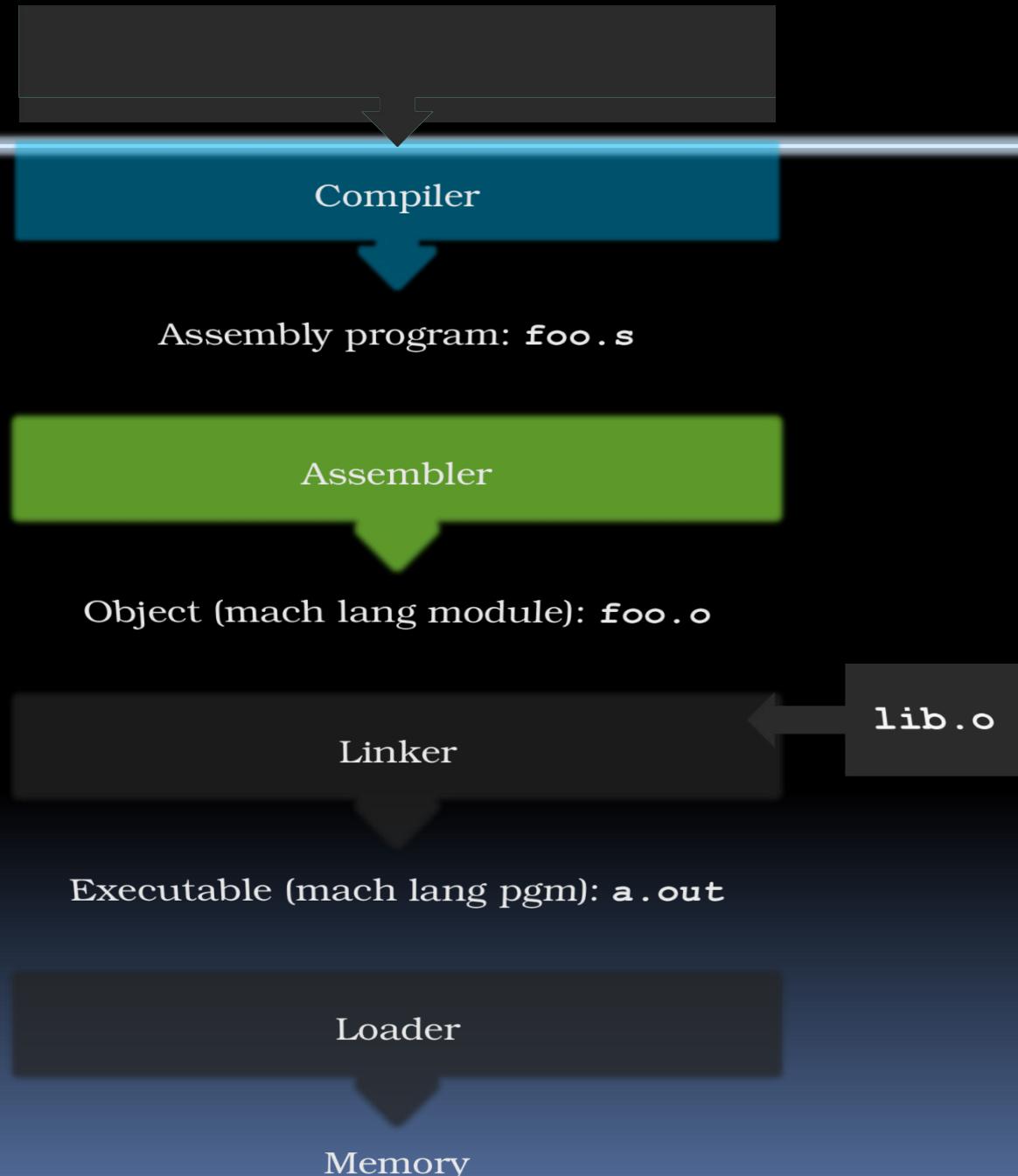
## Lecture 14 – Running a Program II

(Compiling, Assembling, Linking,  
Loading)

Lecturer  
Yuanqing  
Cheng

2020-09-27

# Review



# Symbol Table

---

- List of “items” in this file that may be used by other files.
- What are they?
  - Labels: function calling
  - Data: anything in the .data section; variables which may be accessed across files

# Relocation Table

---

- List of “items” this file needs the address later.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any piece of data connected with an address
    - such as the `la` instruction

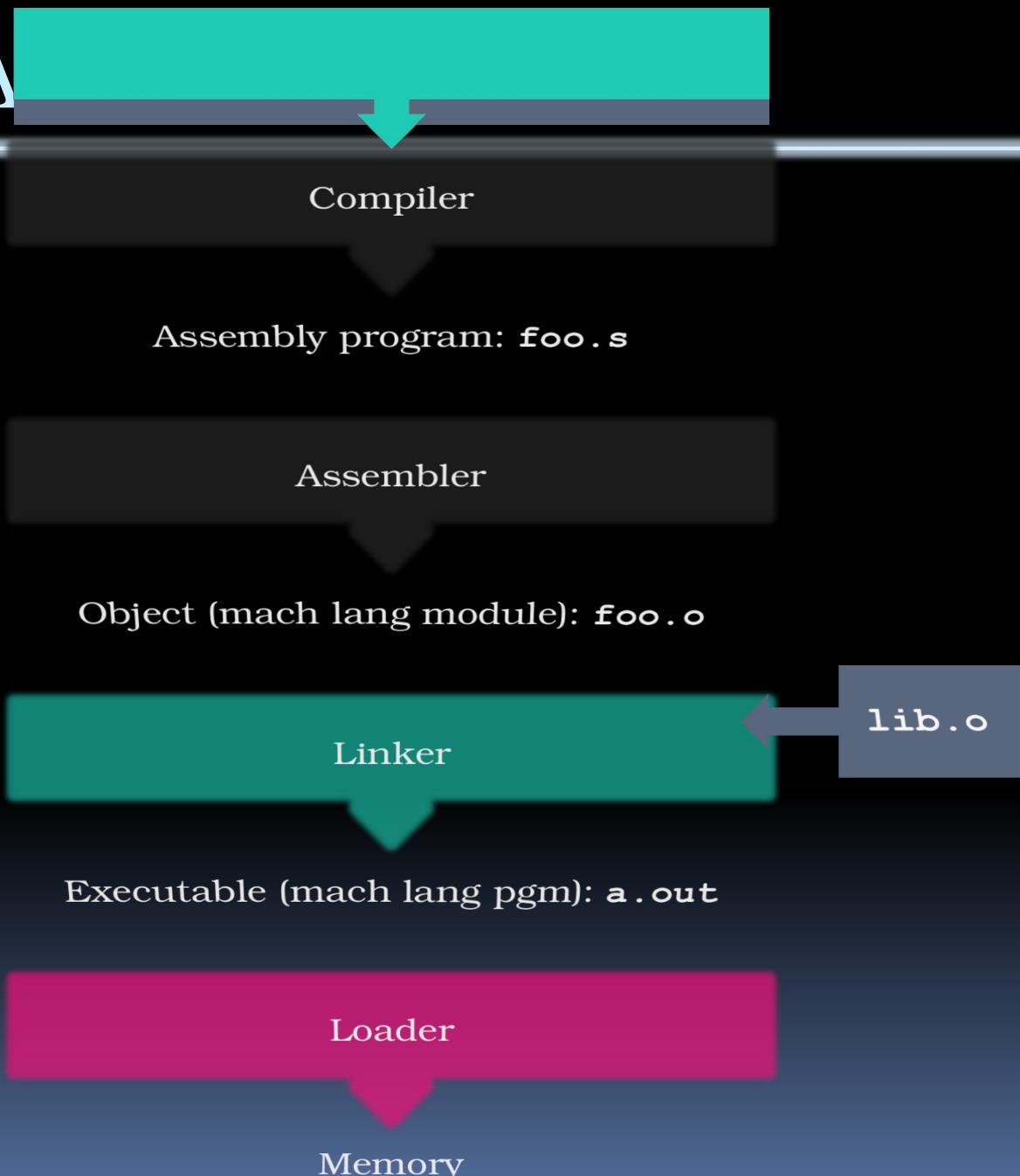
# Object File Format

---

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be “handled”
- symbol table: list of this file’s labels and data that can be referenced
- debugging information
- A standard format is ELF (except MS)

[http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)

# Where A



# Linker (1 / 3)

---

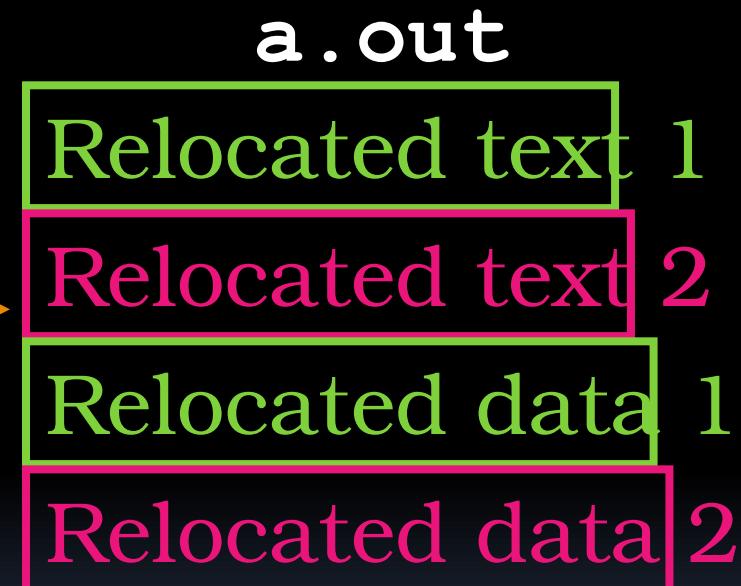
- Input: Object Code files, information tables (e.g., `foo.o`, `libc.o` for MIPS)
- Output: Executable Code (e.g., `a.out` for MIPS)
- Combines several object (`.o`) files into a single executable (“linking”)
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
    - Windows NT source was > 40 M lines of code!
  - Old name “Link Editor” from editing the “links” in jump and link instructions

# Linker (2/3)

.o file 1



.o file 2



# Linker (3/3)

---

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table; handle each entry
  - That is, fill in all **absolute addresses**

# Four Types of Addresses we'll discuss

---

- PC-Relative Addressing (beq, bne)

- never relocate
- Absolute Address (j, jal)
  - always relocate
- External Reference (usually jal)
  - always relocate
- Data Reference (often lui and ori)
  - always relocate

# Absolute Addresses in MIPS

- Which instructions need relocation editing?

j<sup>□</sup> / jal<sup>□</sup> format: jump, jump<sub>xxxxx</sub> and link

- Loads and stores to variables in static area,

lw<sup>relative</sup> / sw<sup>relative</sup> \$gp<sup>global pointer</sup> \$x<sup>pointer</sup> address

beq<sup>What about</sup> / bne<sup>conditional branches?</sup> \$rs<sup>r</sup> \$rt<sup>t</sup> address

- PC-relative addressing preserved even if code moves

# Resolving References (1 / 2)

---

- Linker **assumes** first word of first text segment is at address **0x00000000**.
  - (More later when we study “virtual memory”)
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

# Resolving References (2/2)

---

- To resolve references:
  - search for reference (data or label) in all “user” symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

# Static vs Dynamically linked

---

~~libraries~~, what we've described is the traditional way: **statically-linked** approach

- The library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
- It includes the entire library even if not all of it will be used.
- Executable is self-contained.
- An alternative is **dynamically linked libraries** (DLL), common on Windows & UNIX platforms

# Dynamically linked libraries

---

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file (`libXYZ.so`) upgrades every program that uses library “XYZ”

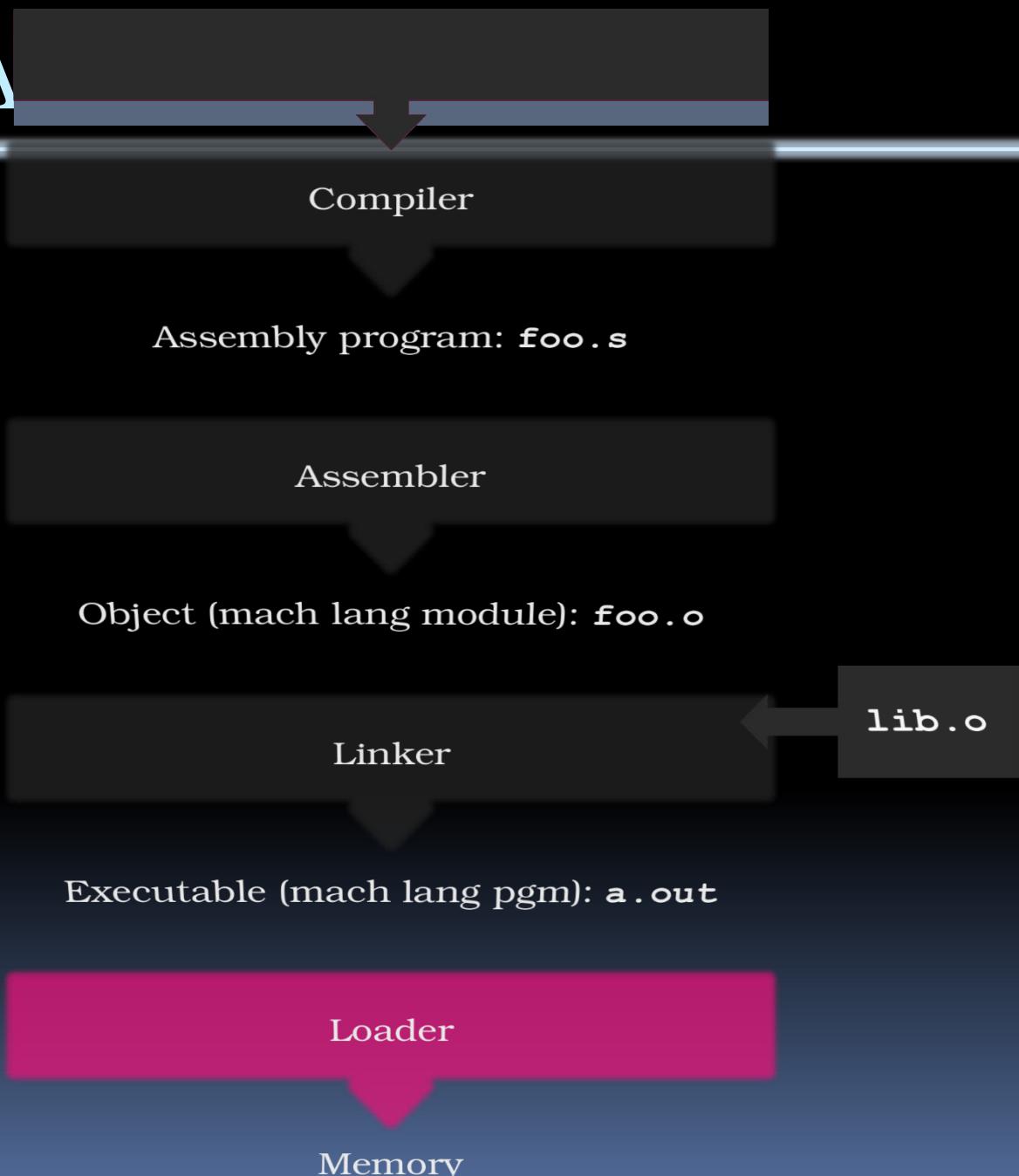
Overall, having the executable on the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.  
any more

# Dynamically linked libraries

---

- The prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  - The linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - This can be described as “linking at the machine code level”
  - This isn’t the only way to do it...

# Where A



# Loader (1 / 3)

---

- Input: Executable Code  
(e.g., a.out for MIPS)
- Output: (program is run)
- Executable files are stored on disk.
- When one is run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

# Loader (2/3)

---

- So what does a loader do?
  - Reads executable file's header to determine size of text and data segments
  - Creates new address space for program large enough to hold text and data segments, along with a stack segment
  - Copies instructions and data from executable file into the new address space

# Loader (3/3)

---

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1<sup>st</sup> free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

# Peer Instruction

---

**Which of the following instr. may need to be edited during link phase?**

Loop: lui \$at, 0xABCD }# 1

ori \$a0,\$at, 0xFEDC

bne \$a0,\$v0, Loop # 2

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

# Peer Instruction Answer

---

**Which of the following instr. may need to be edited during link phase?**

data reference; relocate

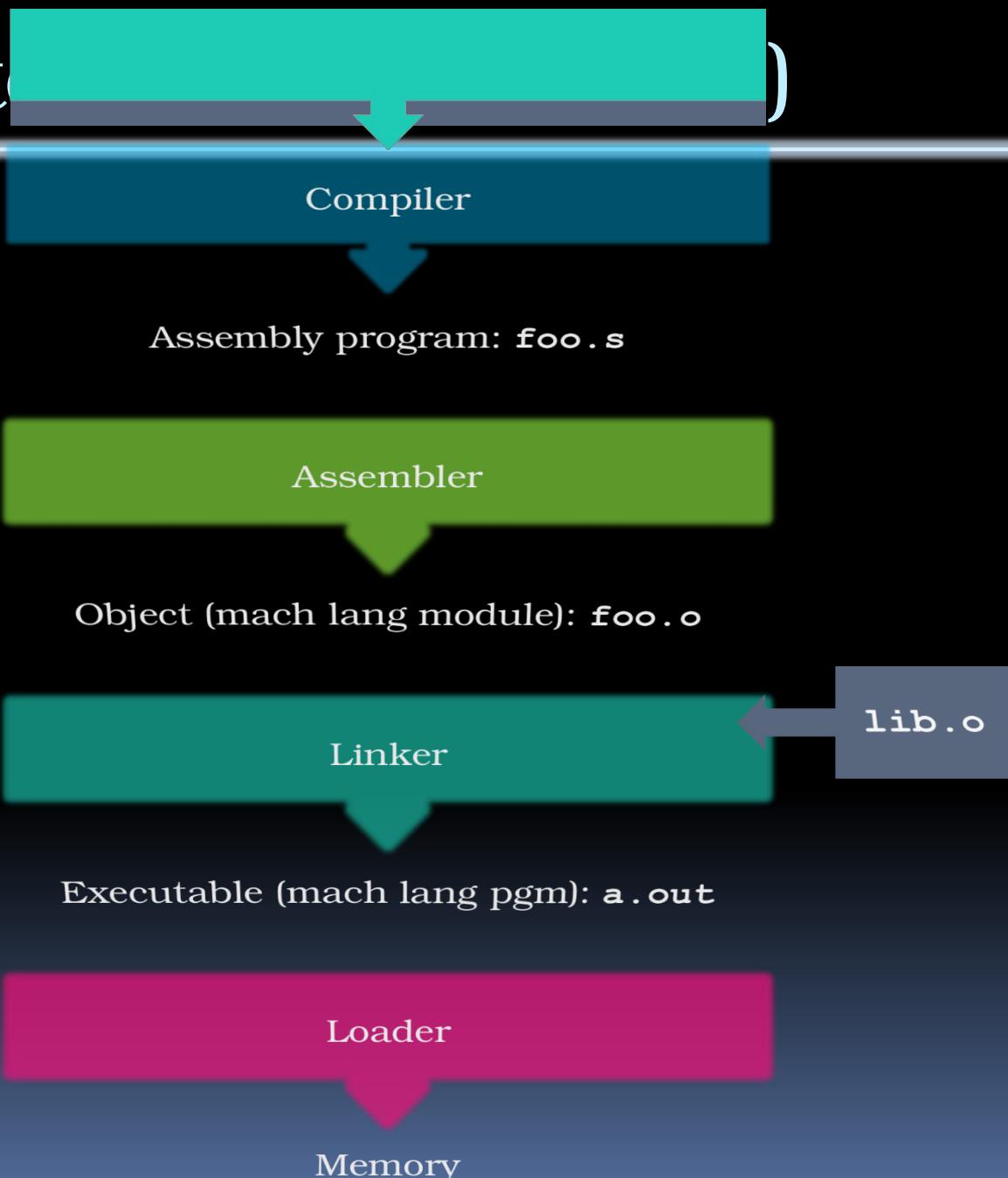
Loop: lui \$at, 0xABCD }# 1

ori \$a0,\$at, 0xFEDC PC-relative branch; OK

bne \$a0,\$v0, Loop # 2

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

# Things to know about the build process



# Things to Remember (2/3)

---

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A .s file becomes a .o file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.

# Things to Remember 3/3

---

- Stored Program concept is very powerful. It means that instructions sometimes act just like data. Therefore we can use programs to manipulate other programs!
  - Compiler      Assembler      Linker (      Loader)

# Bonus slides

---

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus

# Big Endian vs. Little

## Endian

Big-endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- **The order in which BYTES are stored in memory**
- Bits always stored as usual. (E.g., **0xC2=0b 1100 0010**)

Consider the number 1025 as we normally write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

### Big Endian

- ADDR3 ADDR2 ADDR1 ADDR0  
BYTE0 BYTE1 BYTE2 BYTE3  
00000001 00000100 00000000 00000000
- ADDR0 ADDR1 ADDR2 ADDR3  
BYTE3 BYTE2 BYTE1 BYTE0  
00000000 00000000 00000100 00000001

### Little Endian

- ADDR3 ADDR2 ADDR1 ADDR0  
BYTE3 BYTE2 BYTE1 BYTE0  
00000000 00000000 00000100 00000001
- ADDR0 ADDR1 ADDR2 ADDR3  
BYTE0 BYTE1 BYTE2 BYTE3  
00000001 00000100 00000000 00000000

[www.webopedia.com/TERM/b/big\\_endian.html](http://www.webopedia.com/TERM/b/big_endian.html)

[searchnetworking.techtarget.com/sDefinition/0,,sid7\\_gci211659,00.html](http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci211659,00.html)

[www.noveltheory.com/TechPapers/endian.asp](http://www.noveltheory.com/TechPapers/endian.asp)

[en.wikipedia.org/wiki/Big\\_endian](http://en.wikipedia.org/wiki/Big_endian)

Example: C  $\Rightarrow$  Asm  $\Rightarrow$  Obj  $\Rightarrow$  Exe  $\Rightarrow$

---

Run

**C Program Source Code:** *prog.c*

```
#include <stdio.h>

int main (int argc, char *argv[] ) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is
            %d\n",
            sum);
}
```

**“printf” lives in “libc”**

# Compilation: MAL

```
___.text
.align 2
.globl main
main:
    subu $sp,$sp,32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6,$t6
    lw $t8, 24($sp)
    addu $t9,$t8,$t7
    sw $t9, 24($sp)
```

```
addu $t0, $t6, 1
sw $t0, 28($sp)
ble $t0,100, loop
la $a0, str
lw $a1, 24($sp)
jal printf
move $v0, $0
lw $ra, 20($sp)
addiu $sp,$sp,32
jr $ra
.data
.align 0
str:
.asciiiz "The sum
of sq from 0 ..
100 is %d\n"
```

**Where are  
7 pseudo-  
instructions?**

# Compilation: MAL

```
___.text
.align 2
.globl main
main:
    subu $sp,$sp,32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6,$t6
    lw $t8, 24($sp)
    addu $t9,$t8,$t7
    sw $t9, 24($sp)
```

```
addu $t0, $t6, 1
sw $t0, 28($sp)
ble $t0,100, loop
la $a0, str
lw $a1, 24($sp)
jal printf
move $v0, $0
lw $ra, 20($sp)
addiu $sp,$sp,32
jr $ra
.data
.align 0
str:
.asciiiz "The sum
of sq from 0 ..
100 is %d\n"
```

**7 pseudo-instructions underlined**

# Assembly step 1:

Remove pseudoinstructions, assign addresses

00	addiu	\$29, \$29, -32
04	sw	\$31, 20(\$29)
08	sw	\$4, 32(\$29)
0c	sw	\$5, 36(\$29)
10	sw	\$0, 24(\$29)
14	sw	\$0, 28(\$29)
18	lw	\$14, 28(\$29)
1c	multu	\$14, \$14
20	mflo	\$15
24	lw	\$24, 24(\$29)
28	addu	\$25, \$24, \$15
2c	sw	\$25, 24(\$29)

30	addiu	\$8, \$14, 1
34	sw	\$8, 28(\$29)
38	slti	\$1, \$8, 101
3c	bne	\$1, \$0, loop
40	lui	\$4, l.str
44	ori	\$4, \$4, r.str
48	lw	\$5, 24(\$29)
4c	jal	printf
50	add	\$2, \$0, \$0
54	lw	\$31, 20(\$29)
58	addiu	\$29, \$29, 32
5c	jr	\$31

# Assembly step 2

---

## Create relocation table and symbol table

- Symbol Table

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000000	local data

- Relocation Information

Address	Instr.	type
Dependency	0x00000040	lui
	l.str	
0x00000044	ori	r.str
0x0000004c	jal	printf

# Assembly step 3

## Resolve local PC-relative labels

00	addiu	\$29, \$29, -32
04	sw	\$31, 20(\$29)
08	sw	\$4, 32(\$29)
0C	sw	\$5, 36(\$29)
10	sw	\$0, 24(\$29)
14	sw	\$0, 28(\$29)
18	lw	\$14, 28(\$29)
1C	multu	\$14, \$14
20	mflo	\$15
24	lw	\$24, 24(\$29)
28	addu	\$25, \$24, \$15
2C	sw	\$25, 24(\$29)

30	addiu	\$8, \$14, 1
34	sw	\$8, 28(\$29)
38	slti	\$1, \$8, 101
3C	bne	\$1, \$0, -10
40	lui	\$4, <u>l.str</u>
44	ori	\$4, \$4, <u>r.str</u>
48	lw	\$5, 24(\$29)
4C	jal	<u>printf</u>
50	add	\$2, \$0, \$0
54	lw	\$31, 20(\$29)
58	addiu	\$29, \$29, 32
5C	jr	\$31

# Assembly step 4

---

- Generate object (.o) file:
  - Output binary representation for
    - text segment (instructions),
    - data segment (data),
    - symbol and relocation tables.
  - Using dummy “placeholders” for unresolved absolute and external references.

# Text segment in object

file

0x000000	001001110111101111111111111100000
0x000004	101011110111110000000000010100
0x000008	1010111101001000000000000100000
0x00000c	1010111101001010000000000100100
0x000010	1010111101000000000000000110000
0x000014	1010111101000000000000000111000
0x000018	1000111110101100000000000111000
0x00001c	1000111110111000000000000110000
0x000020	0000000111001110000000000110001
0x000024	0010010111001000000000000000001
0x000028	0010100100000010000000001100101
0x00002c	1010111110101000000000000111000
0x000030	00000000000000000111100000010010
0x000034	00000011000011111100100000100001
0x000038	000101000010000011111111110111
0x00003c	1010111110111001000000000011000
0x000040	0011110000000100000000000000000
0x000044	1000111110100101000000000000000
0x000048	000011000001000000000000011101100
0x00004c	0010010000000000000000000000000
0x000050	10001111101111110000000000010100
0x000054	00100111101111010000000000100000
0x000058	00000011111000000000000000000001000
0x00005c	0000000000000000000000000000000100001

# Link step 1: combine `prog.o`,

## `.libc.o`

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables
- Symbol Table
  - Label Address

Label	Address
main:	0x00000000
loop:	0x00000018
str:	0x10000430
printf:	0x000003b0
	...

- Relocation Information

Address	Instr.	Type	Dependency
0x00000040	lui		l.str
0x00000044	ori		r.str
0x0000004c	jal		printf

# Link step 2:

- Edit Addresses in relocation table
  - (shown in TAL for clarity, but done in binary )

00	addiu	\$29, \$29, -32	30	addiu	\$8, \$14, 1
04	sw	\$31, 20 (\$29)	34	sw	\$8, 28 (\$29)
08	sw	\$4, 32 (\$29)	38	slti	\$1, \$8, 101
0c	sw	\$5, 36 (\$29)	3c	bne	\$1, \$0, -10
10	sw	\$0, 24 (\$29)	40	lui	\$4, <u>4096</u>
14	sw	\$0, 28 (\$29)	44	ori	\$4, \$4, <u>1072</u>
18	lw	\$14, 28 (\$29)	48	lw	\$5, 24 (\$29)
1c	multu	\$14, \$14	4c	jal	<u>812</u>
20	mflo	\$15	50	add	\$2, \$0, \$0
24	lw	\$24, 24 (\$29)	54	lw	\$31, 20 (\$29)
28	addu	\$25, \$24, \$15	58	addiu	\$29, \$29, 32
2c	sw	\$25, 24 (\$29)	5c	jr	\$31

# Link step 3:

---

- Output executable of merged modules.
  - Single text (instruction) segment
  - Single data segment
  - Header detailing size of each segment
- NOTE:
  - The preceding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.