Machine Learning's Slides

Author: Pannenets.F

Date: September 21, 2020

Je reviendrai et je serai des millions. «Spartacus»

Part I

part

Chapter 1 Week1: Intro

1.1 Introduction

Computer Architecture
（计算机体系结构）

Lecture #1 – Introduction

Lecturer Yuanqing Cheng （成元庆）

School of Microelectronics

Beihang University

www.cadetlab.cn

“I stand on the shoulders of giants…”

Prof David Patterson

Prof Dan Garcia

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

What are prerequisites of this course ?

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

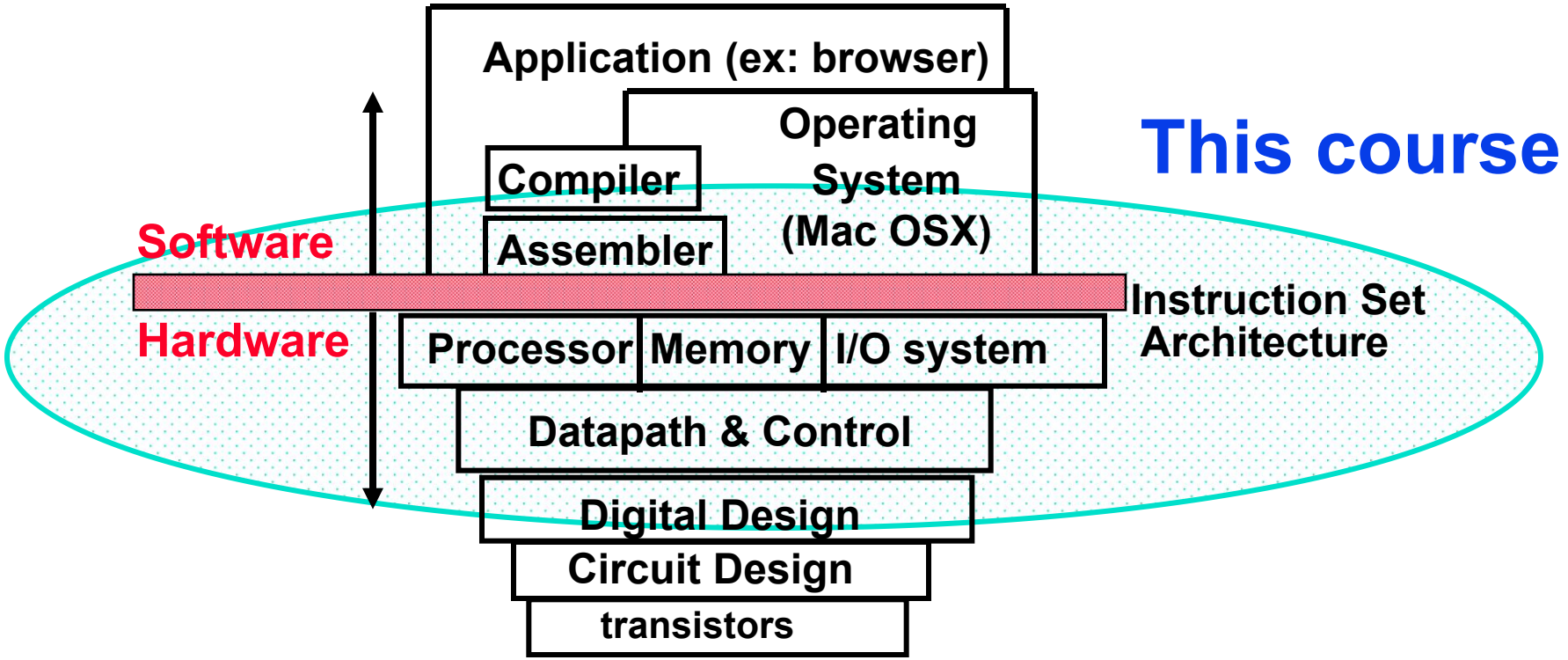
Are Computers Smart?

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

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

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

Compiler

Assembly Language Program (e.g.,MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

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

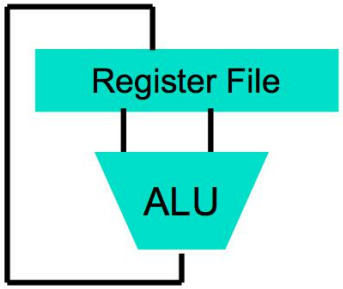
Architecture Implementation

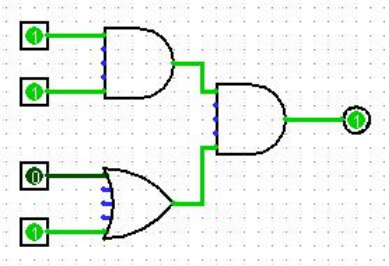
Logic Circuit Description (Circuit Schematic Diagrams)

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

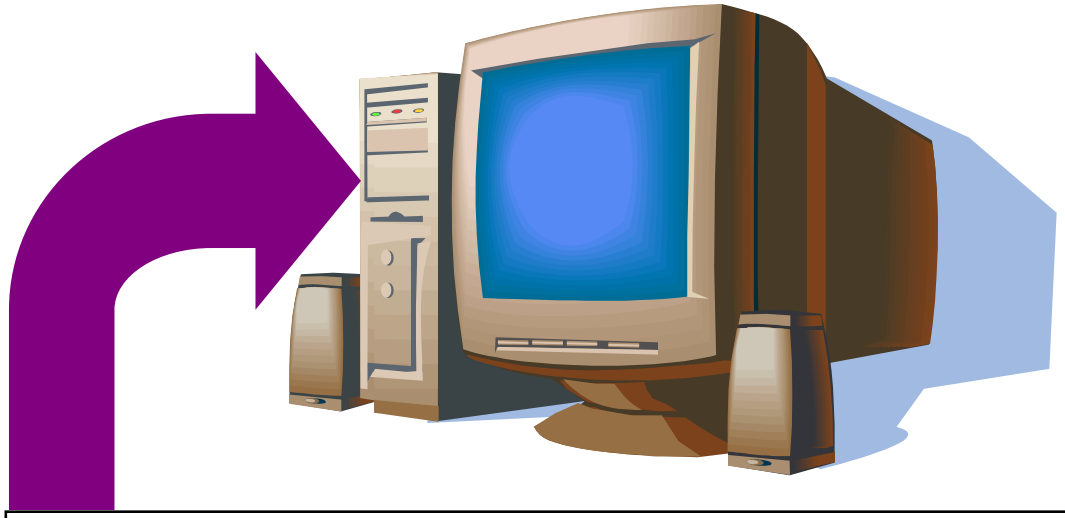
```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

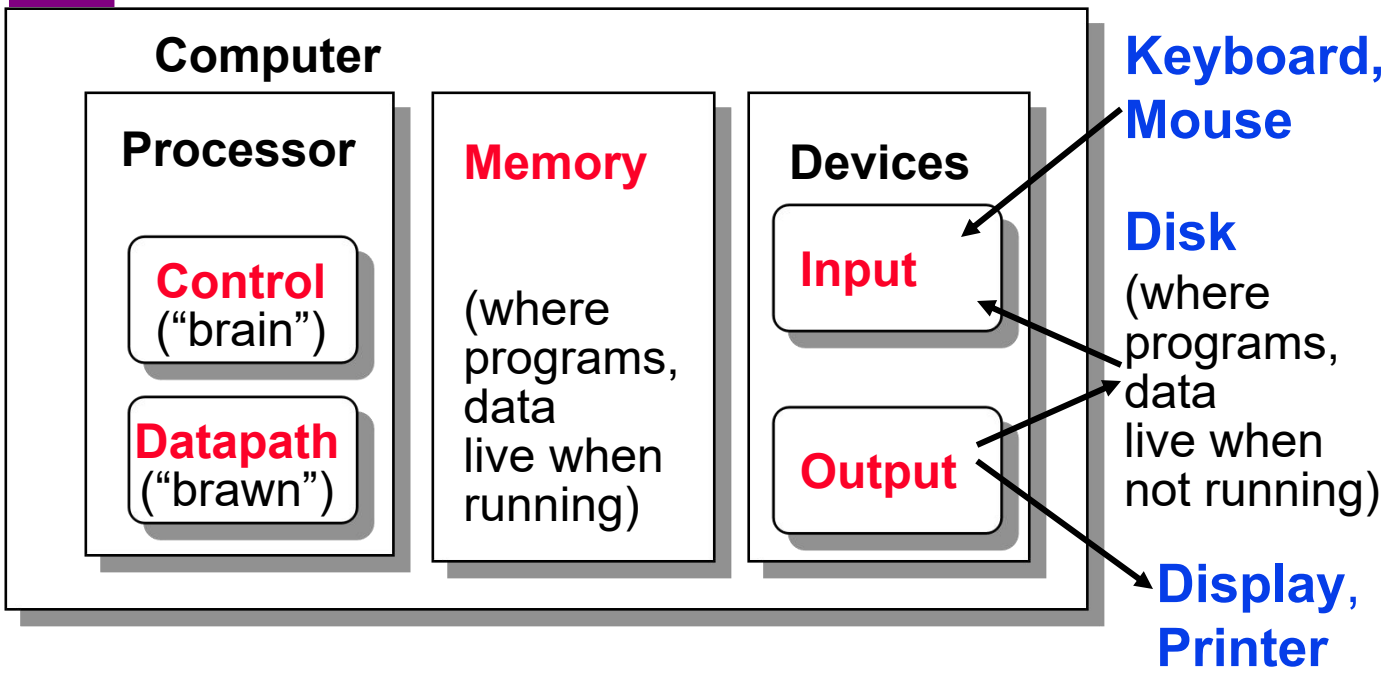
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





Anatomy: 5 components of any Computer





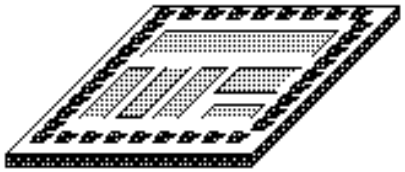
Overview of Physical Implementations

The hardware out of which we make systems.

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

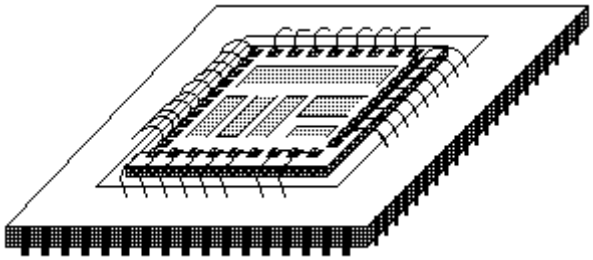
Integrated Circuits (2020 state-of-the-art)

Bare Die



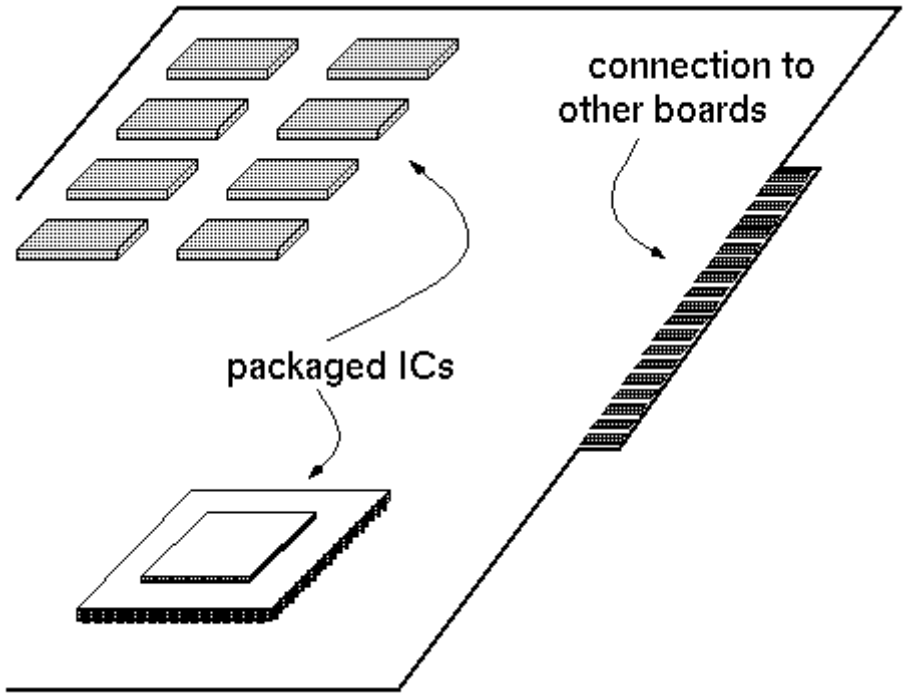
- Primarily Crystalline Silicon
- 1mm - 25mm on a side
- feature size ~ 14/7 nm
- Billions of transistors
- (25 - 100M “logic gates”)
- 3 - 12 conductive layers
- “CMOS” (complementary metal oxide semiconductor) - most common.

Chip in Package



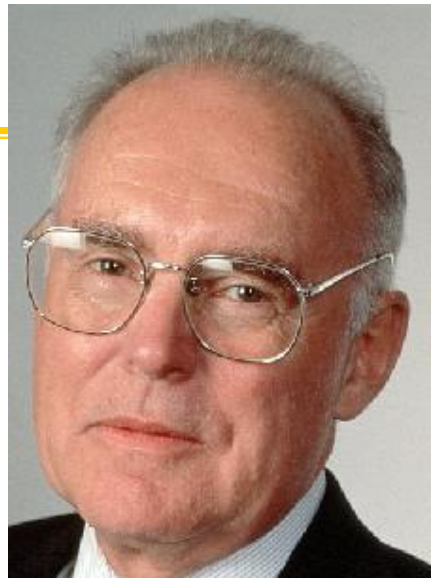
- Package provides:
  - spreading of chip-level signal paths to board-level
  - heat dissipation.
- Ceramic or plastic with gold wires.

Printed Circuit Boards

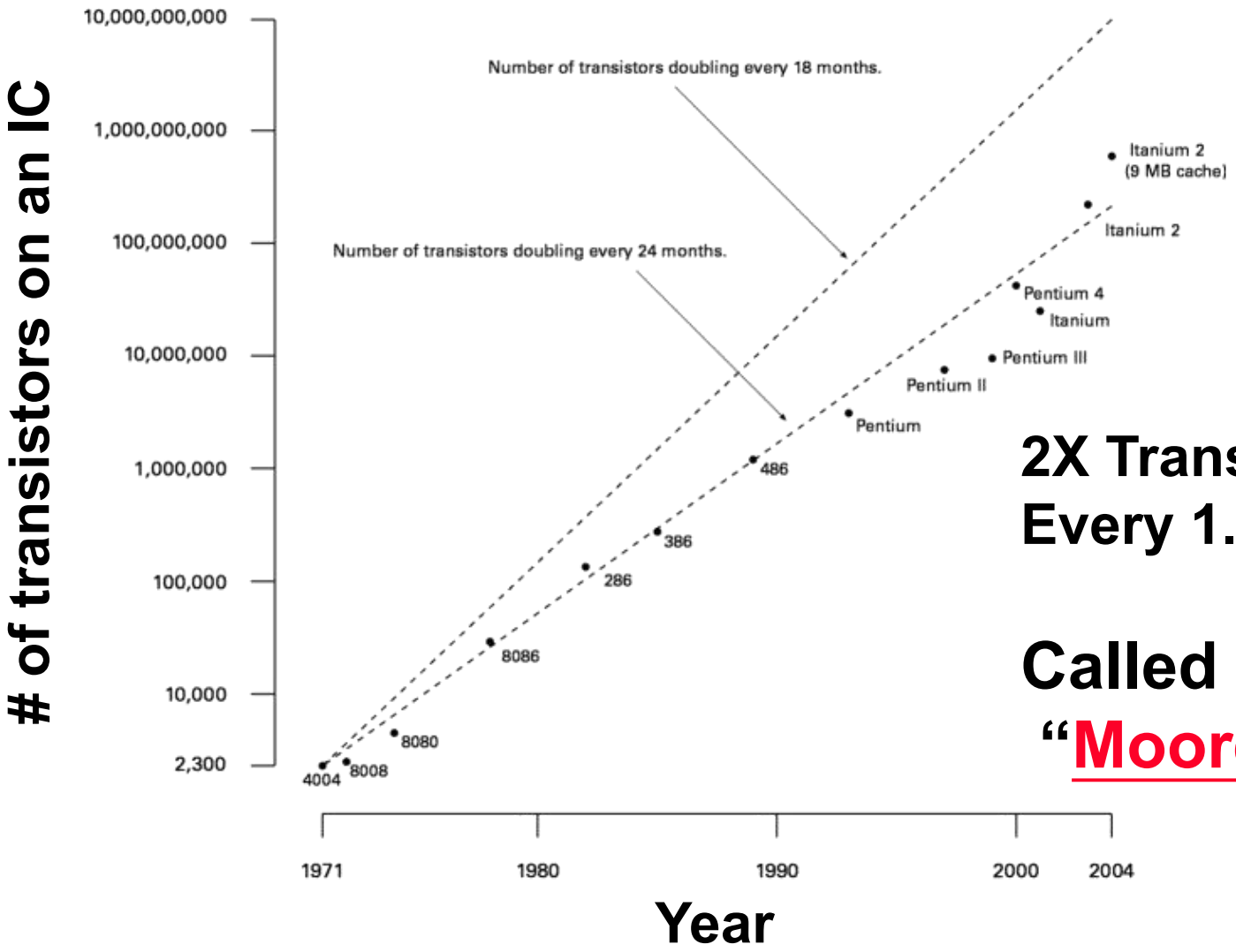


- fiberglass or ceramic
- 1-20 conductive layers
- 1-20 in on a side
- IC packages are soldered down.
- Provides:
  - Mechanical support
  - Distribution of power and heat.

Technology Trends:
Microprocessor Complexity



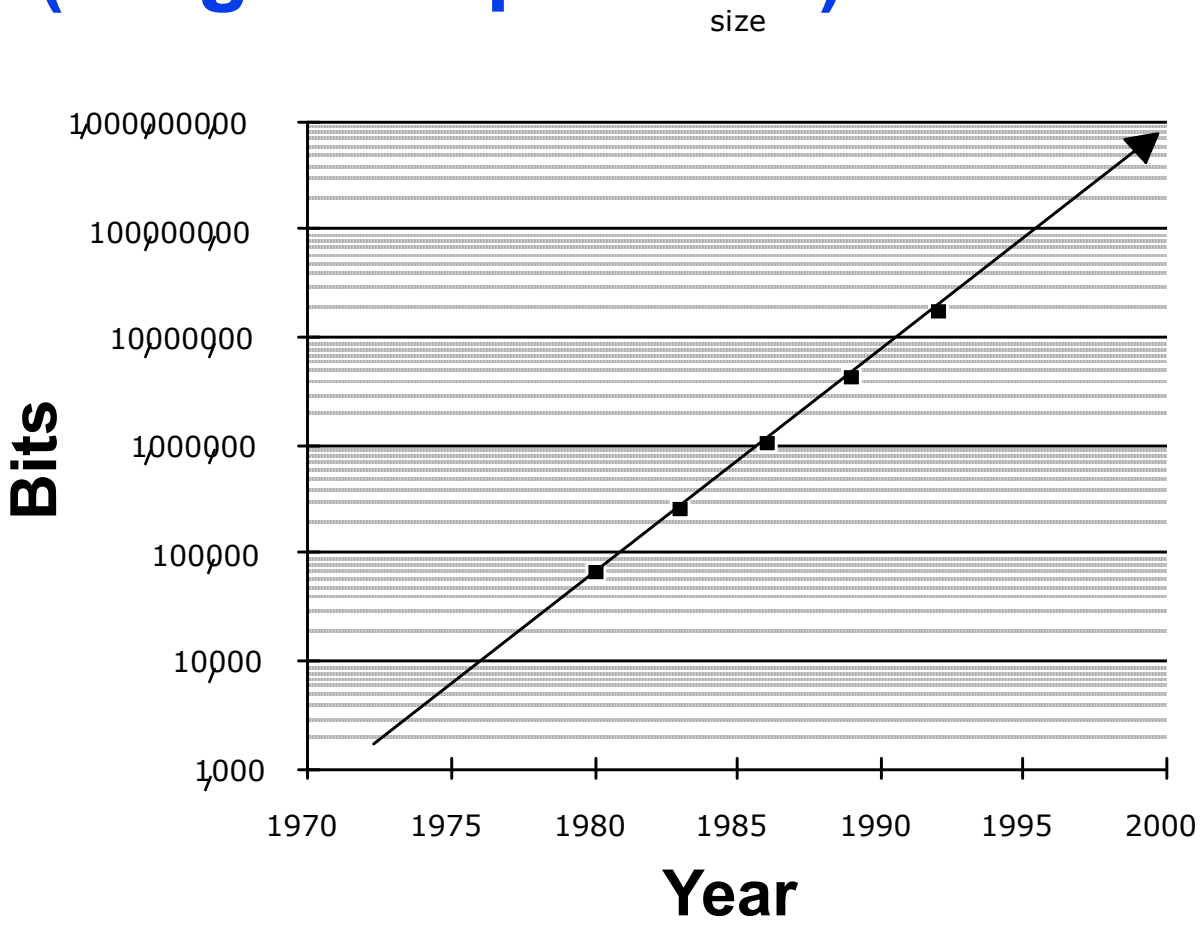
Gordon Moore
Intel Cofounder!



2X Transistors / Chip
Every 1.5 years

Called
"Moore's Law"

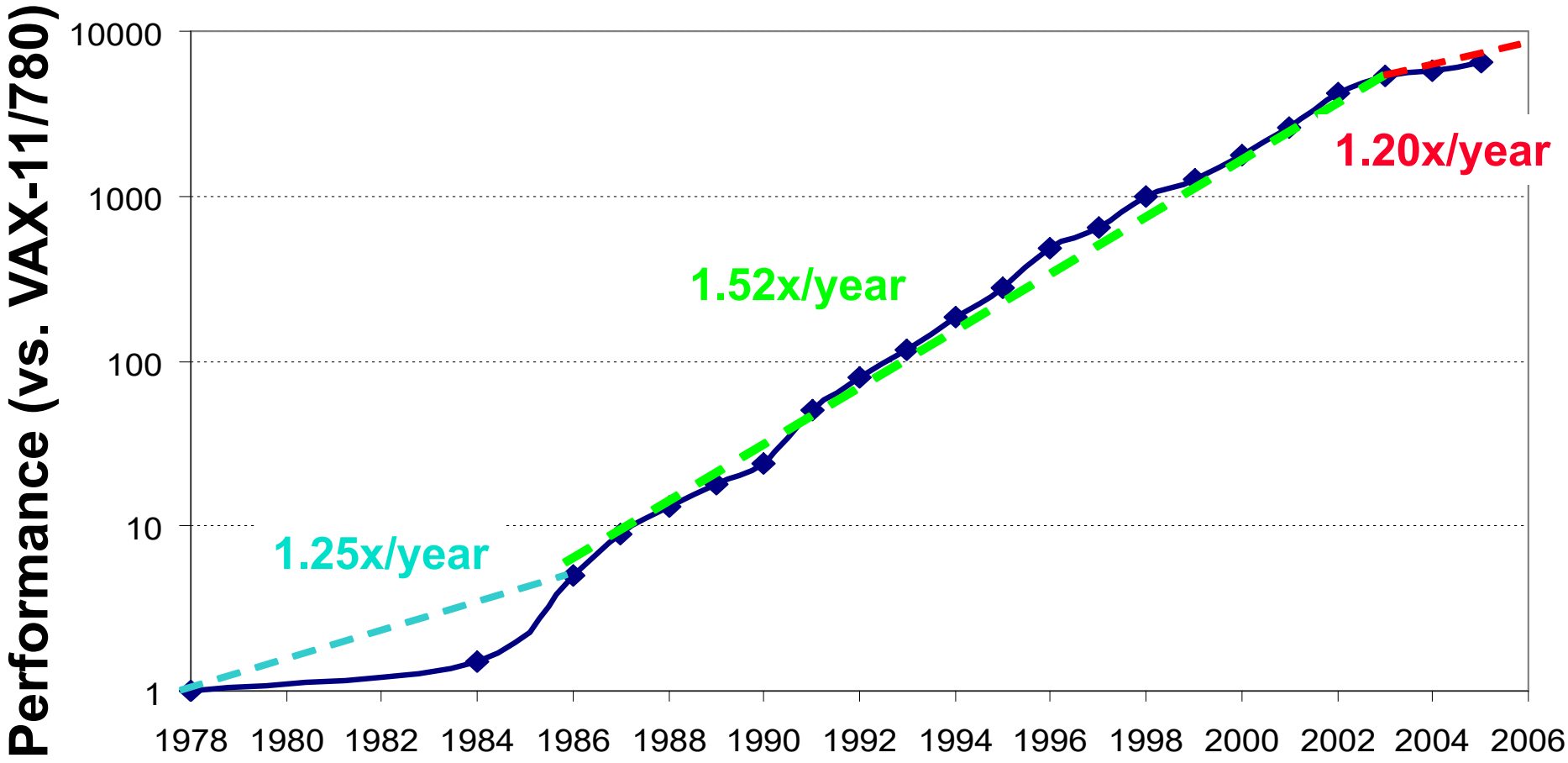
Technology Trends: Memory Capacity (Single-Chip DRAM)



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

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

Technology Trends: Uniprocessor Performance (SPECint)



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

Computer Technology - Dramatic Change!

- **Memory**
  - **DRAM capacity: 2x / 2 years (since ‘96); 64x size improvement in last decade.**
- **Processor**
  - **Speed 2x / 1.5 years (since ‘85); [slowing!] Now almost remain the same.**
- **Disk**
  - **Capacity: 2x / 1 year (since ‘97) 250X size in last decade.**

Computer Technology - Dramatic Change!

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

- **State-of-the-art PC : (at least…)**
  - **Processor clock speed:** 4,000 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

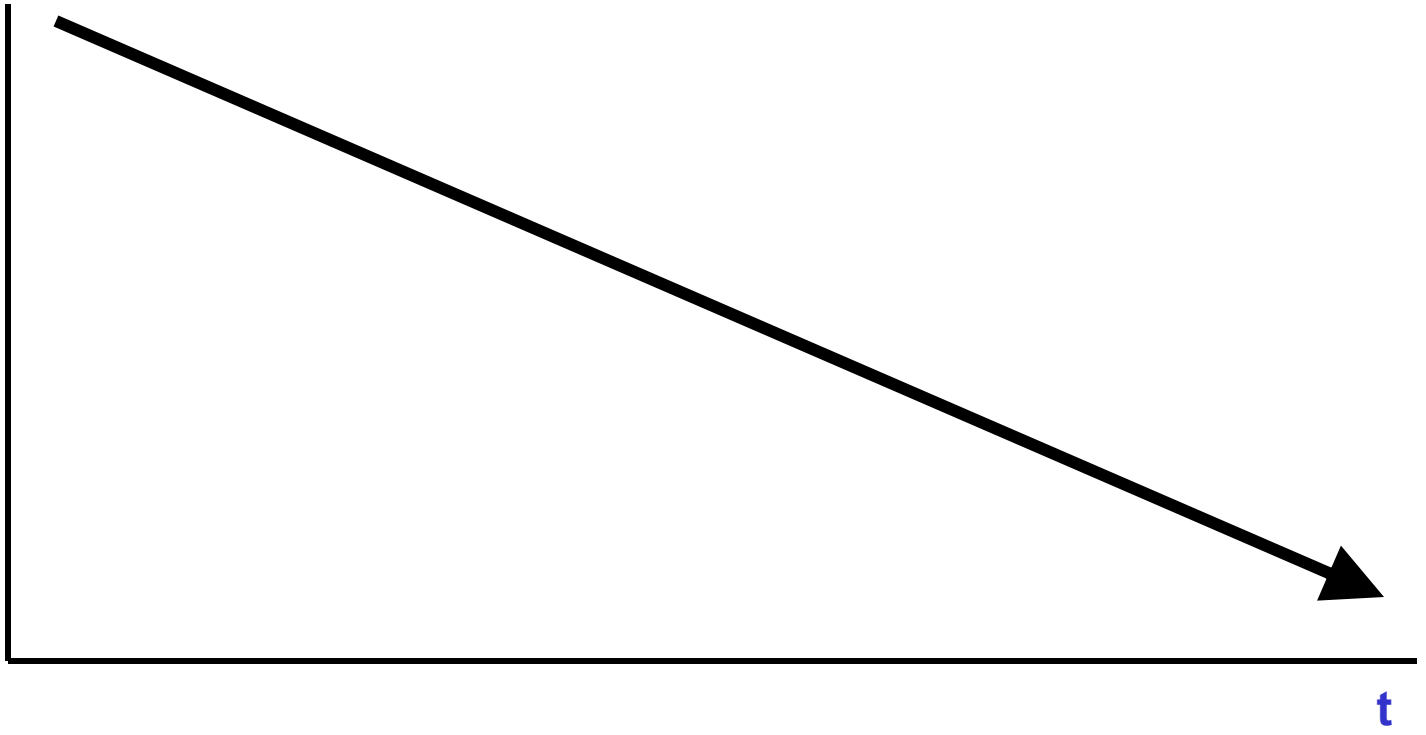
- **Enhance C programming skill**
  - If you know one, you should be able to learn another programming language largely on your own
  - If you know C++ or Java, it should be easy to pick up their ancestor, C
- **Assembly Language Programming**
  - This is a skill you will pick up, as a side effect of 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…”

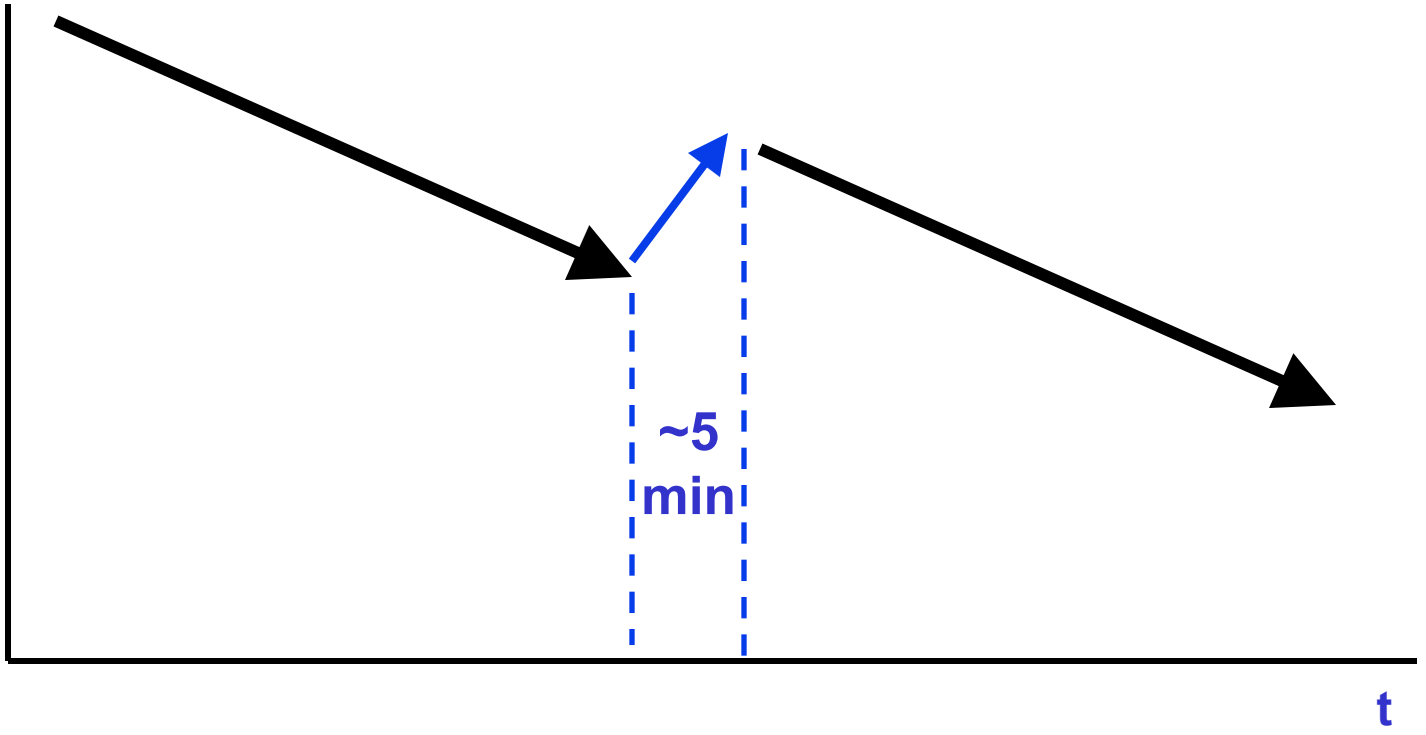
Our schedule may change slightly depending on some factors.
This includes lectures, assignments & labs…

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!

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

Course Problems…Cheating

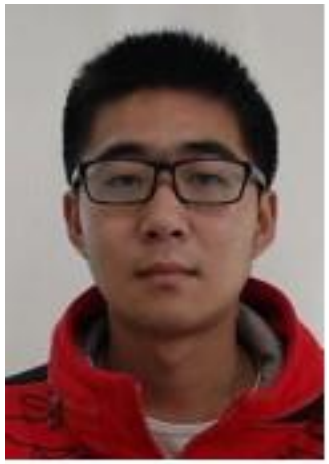
- What is cheating?
  - Studying together in groups is encouraged.
  - Turned-in work must be *completely* your own.
  - Common examples of cheating: running out of time on a assignment and then pick up output, take homework from box and copy, person asks to borrow solution "just to take a look", copying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

My goal as an instructor

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

Teaching Assistants

- Jiacheng Ni



Summary

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

Control Datapath Memory Input Output

Processor

Reference slides

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

Course Lecture Outline

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

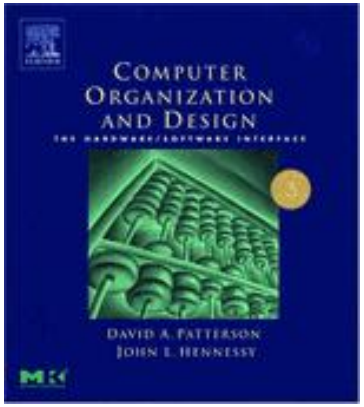
Homeworks and Projects

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

Your final grade

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

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）

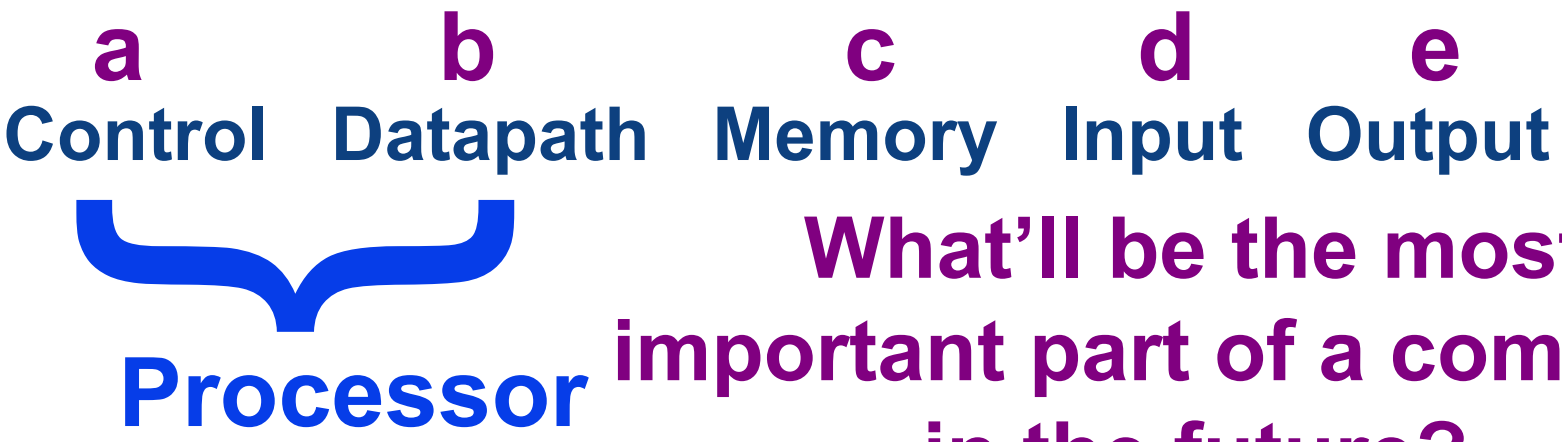
www.cadetlab.cn

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



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

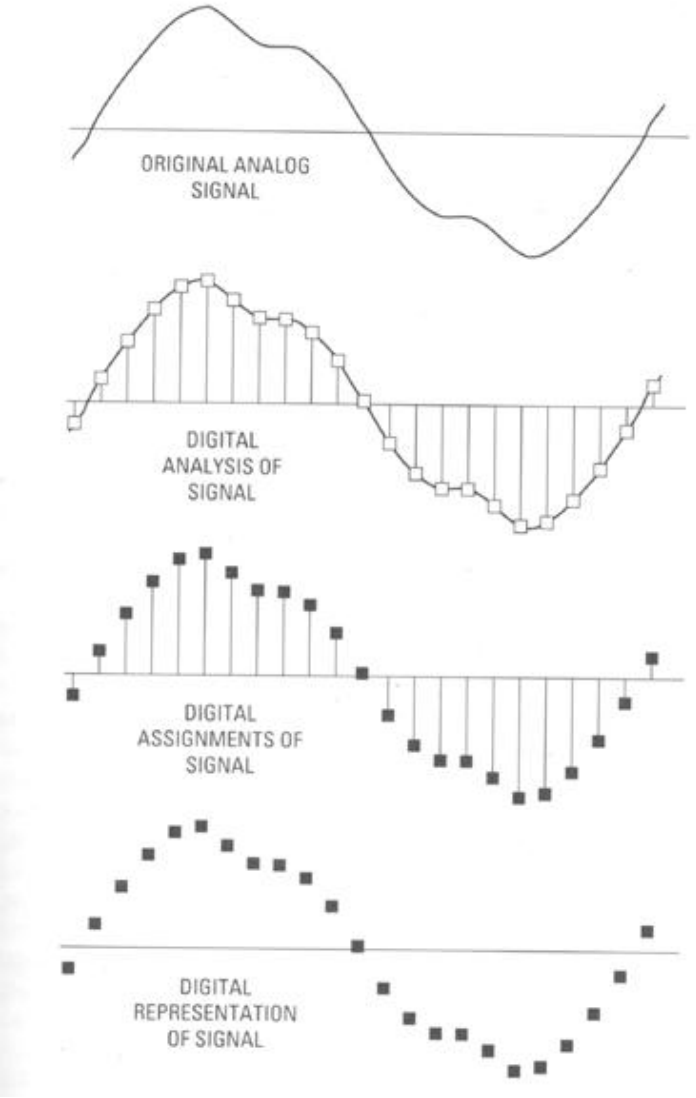
Putting it all in perspective…

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

– Robert X. Cringely

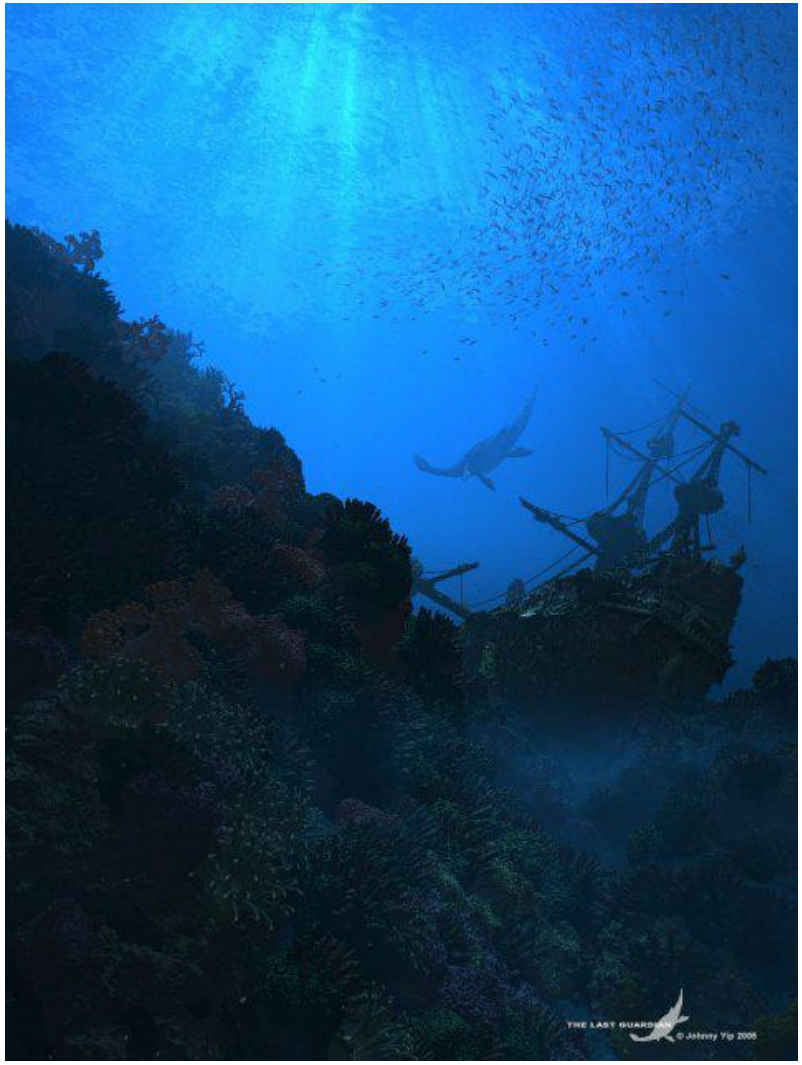
Data input: Analog ➔ Digital

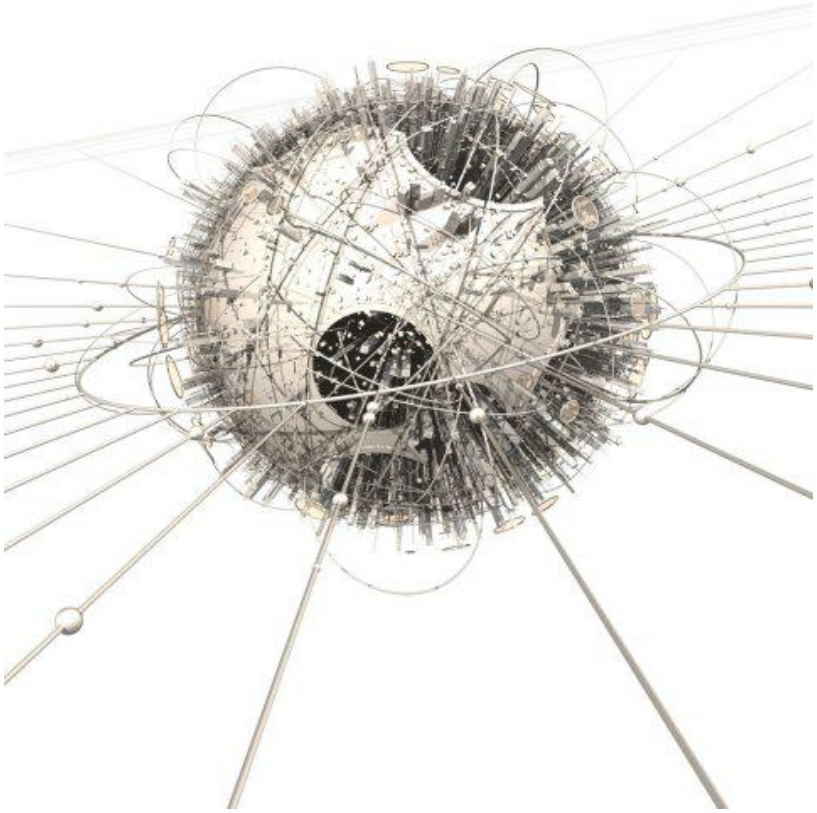
- Real world is analog!
- To import analog information, we must do two things
  - Sample
    - E.g., for a CD, every 44,100ths of a second, we ask a music signal how loud it is.
  - Quantize
    - For every one of these samples, we figure out where, on a 16-bit (65,536 tic-mark) "yardstick", it lies.



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

Digital data not nec born Analog…





hof.povray.org

BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters ⇒ 5 bits ($2^5 = 32$)
  - upper/lower case + punctuation ⇒ 7 bits (in 8) ("ASCII")
  - standard code to cover all the world's languages ⇒ 8,16,32 bits ("Unicode") `www.unicode.com`

- Logical values?
  - 0 ⇒ False, 1 ⇒ True
- colors ? Ex: *Red (00)* *Green (01)* *Blue (11)*
- locations / addresses? commands?
- MEMORIZE: N bits ⇔ at most 2^N things

How many bits to represent π ?

a) 1

b) 9 (π = 3.14, so that's 011 "." 001 100)

c) 64 (Since Macs are 64-bit machines)

d) Every bit the machine has!

e) ∞

What to do with representations of numbers?

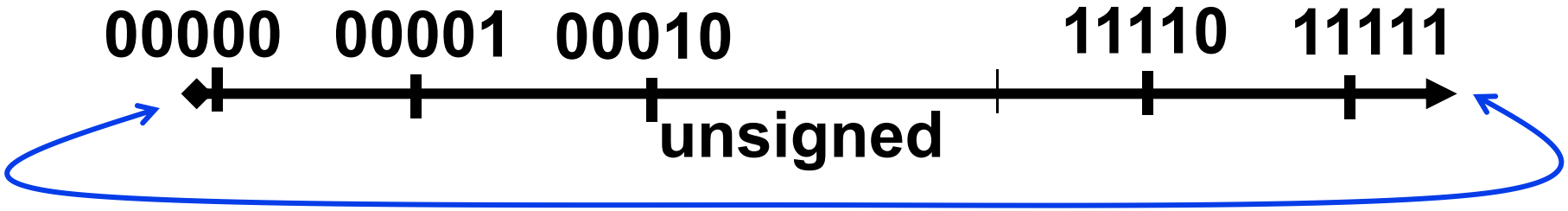
- **Just what we do with numbers!**
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them

```
    1   1
    1   0   1   0
+   0   1   1   1
-------------------
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 ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.

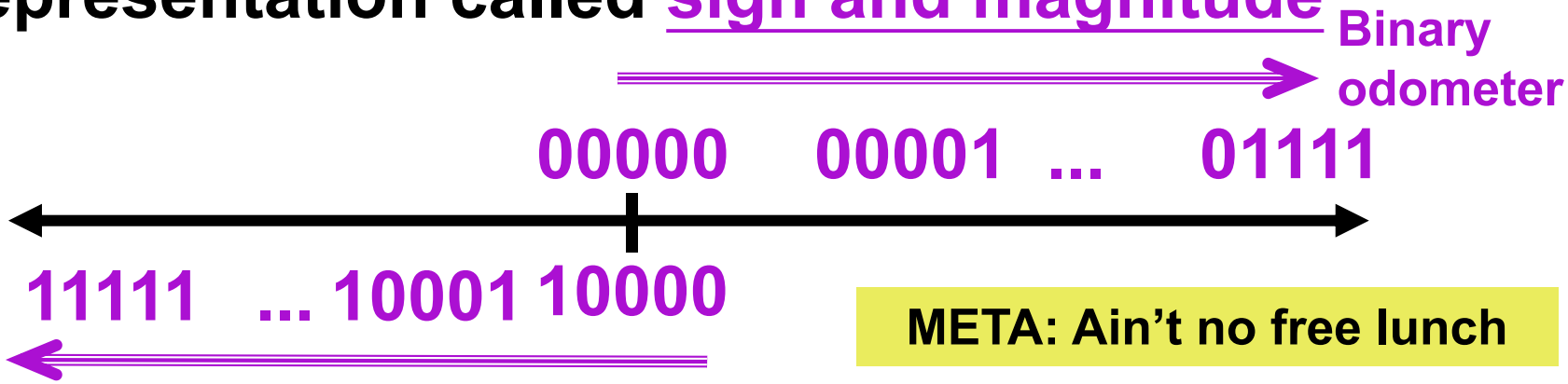


How to Represent Negative Numbers?

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

- So far, unsigned numbers

- Obvious solution: define leftmost bit to be sign!
  - 0 ➔ + 1 ➔ –
  - Rest of bits can be numerical value of number
- Representation called sign and magnitude



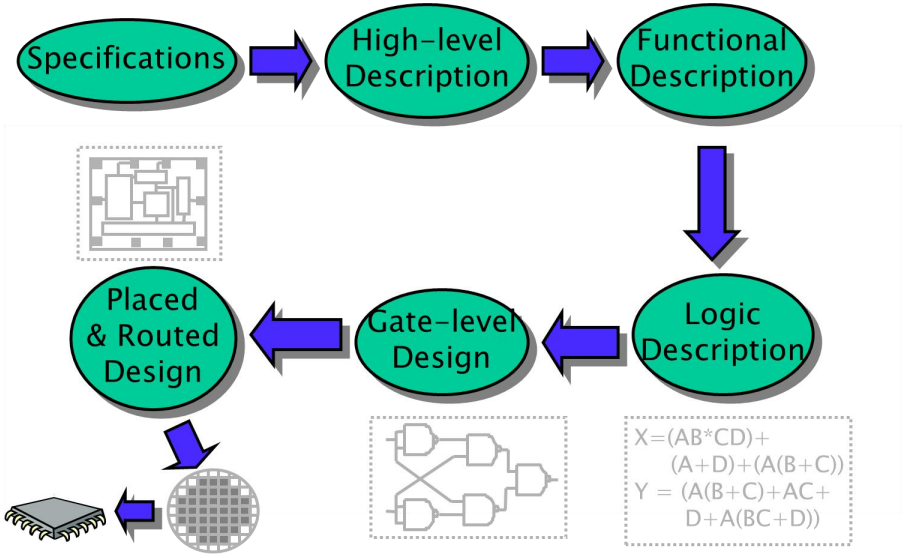
META: Ain't no free lunch

Shortcomings of sign and magnitude?

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

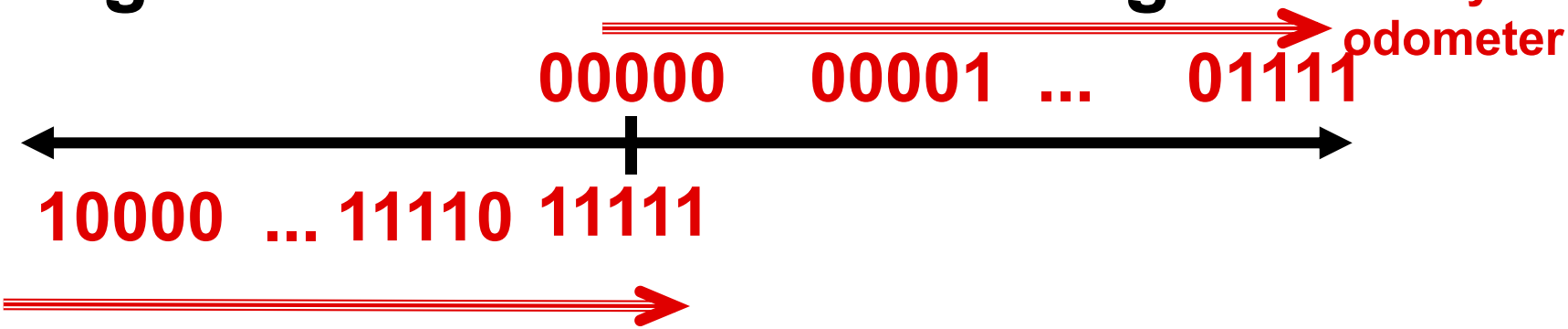
Great EDA course I supervise

- **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` = $+0_{ten}$
  - `0xFFFFFFFF` = -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 ≥ 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:

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

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

$= 1x-(2^3) + 1x2^2 + 0x2^1 + 1x2^0$

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

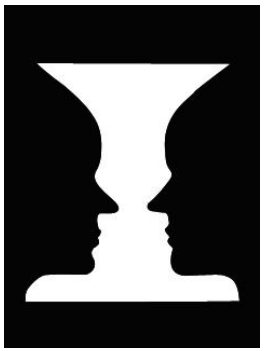
$= -8 + 4 + 0 + 1$

$= -8 + 5$

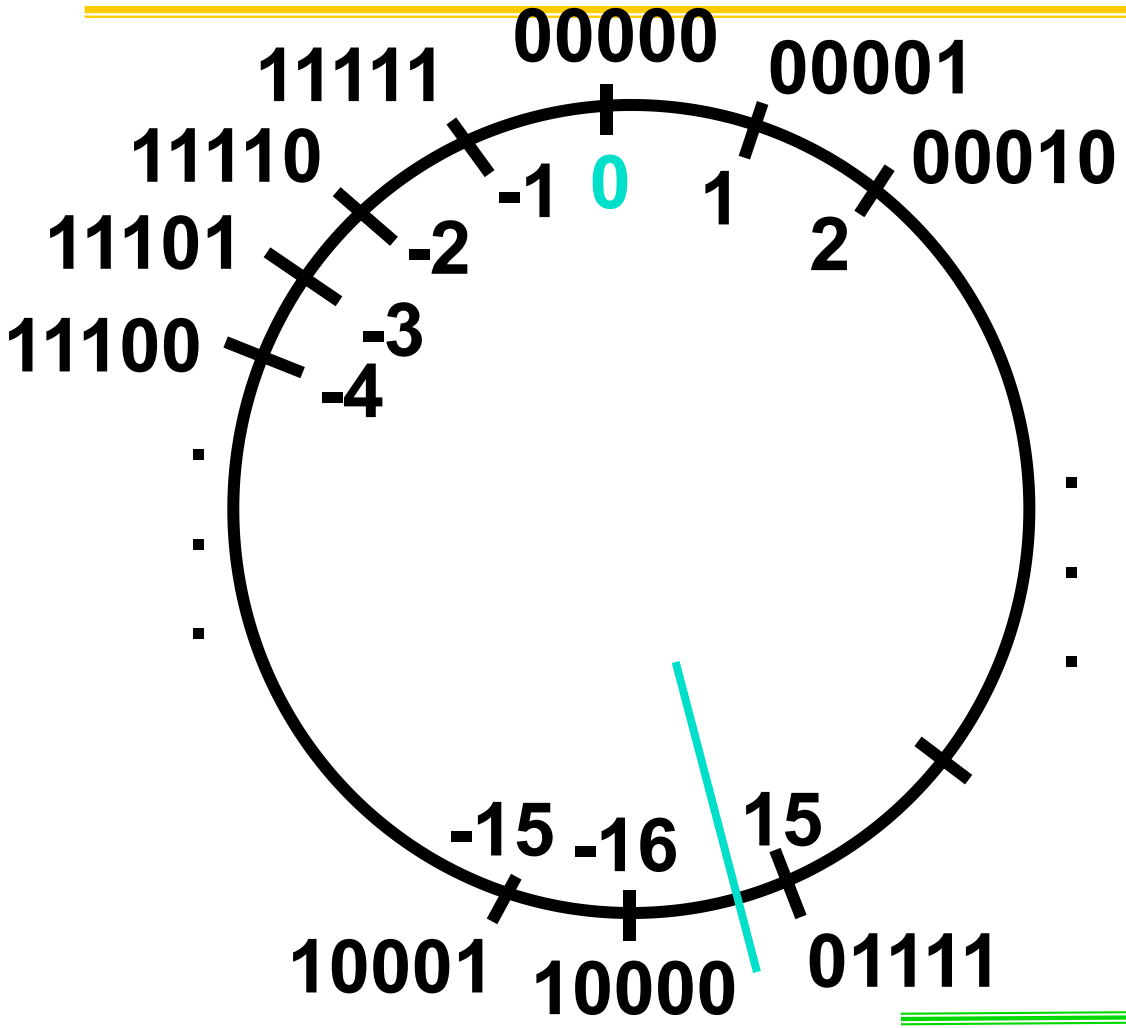
$= -3_{ten}$

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

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



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



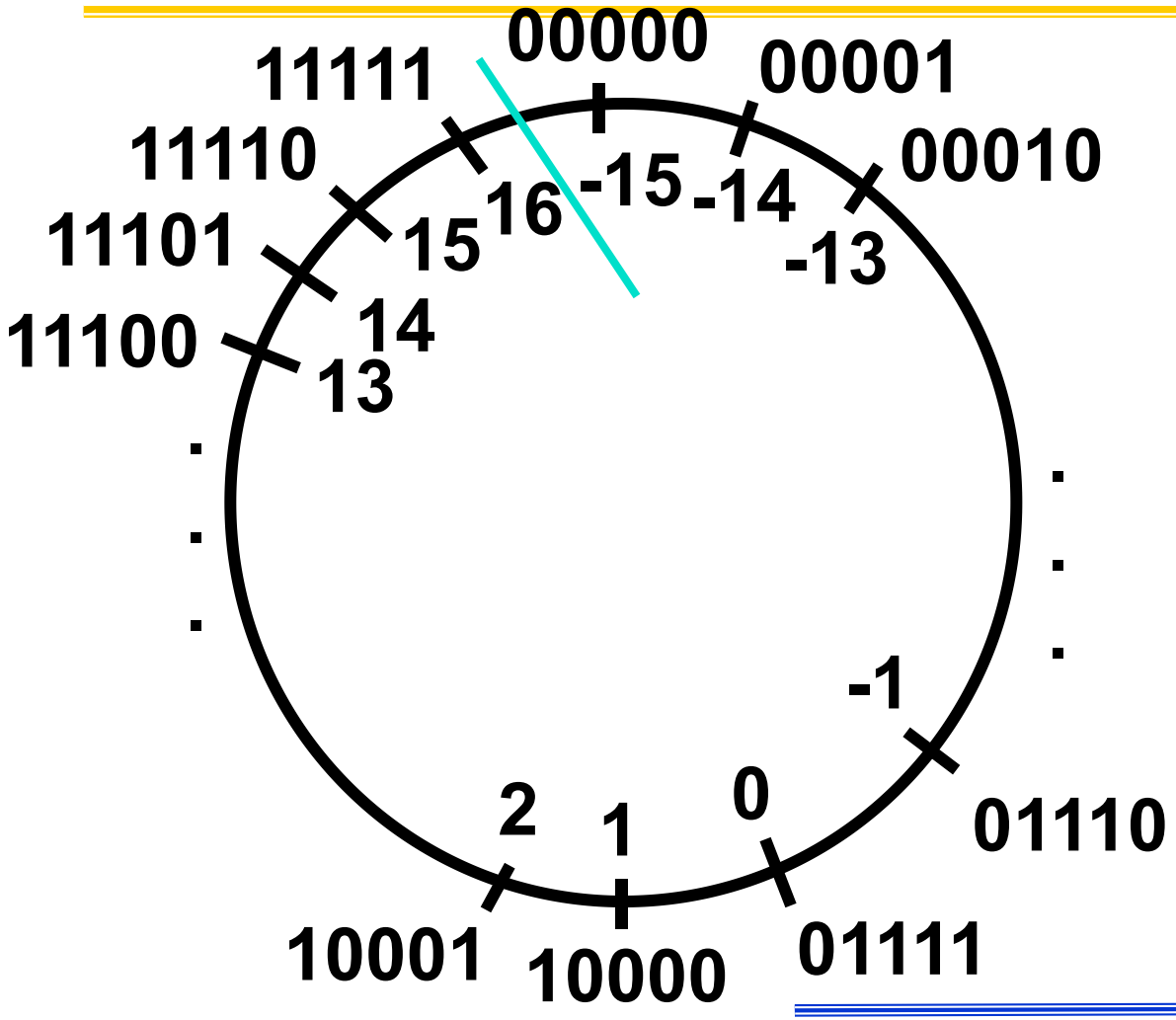
- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?

Binary odometer

00000 00001 ... 01111

10000 ... 11110 11111

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



- # = unsigned + bias
- Bias for N bits chosen as $-(2^{N-1}-1)$
- one zero
- how many positives?

01111 10000 ... 11110 11111

00000 00001 ... 01110

Binary odometer

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

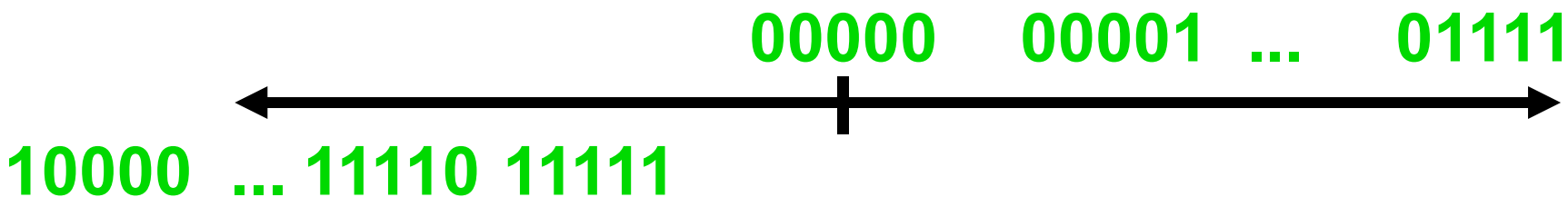
Shifting binary point means "divide number by some power of 2. E.g., $11_{10} = 1011.0_2 \rightarrow 10.110_2 = (11/4)_{10} = 2.75_{10}$

And in summary…

META: We often make design decisions to make HW simple

- We represent "things" in computers as particular bit patterns: N bits ⇒ 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 ∞ ; 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:
  - $32_{ten} == 32_{10} == 0x20 == 100000_2 == 0b100000$
  - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing

Two's Complement for N=32

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

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

Two's comp. shortcut: Sign extension

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

$$1111\ 1111\ 1111\ 1100_{two}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$$

Chapter 2 Week2: MIPS

2.1 MIPS Arithmetic

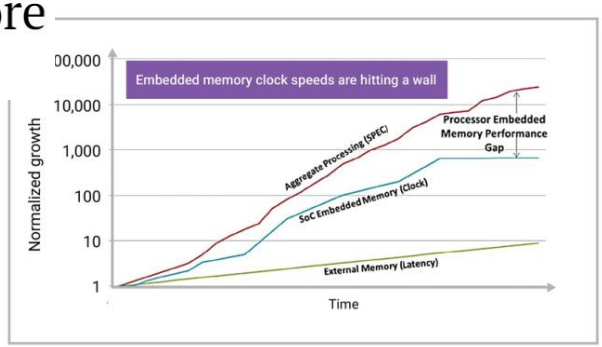
Computer Architecture
（计算机体系结构）

Lecture #3 – Introduction to MIPS Assembly language : Arithmetic

Lecturer Yuanqing Cheng （成元庆）

www.cadetlab.cn

Meeting Increasing Performance Requirements in Embedded Applications with Scalable Multicore Processors

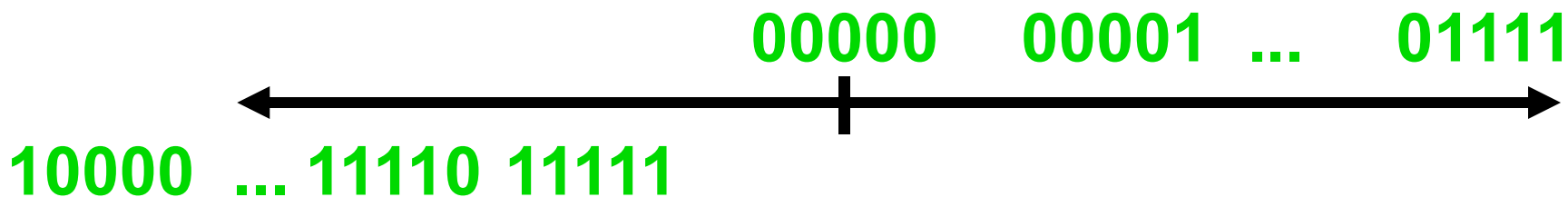


Review

META: We often make design decisions to make HW simple

- We represent "things" in computers as particular bit patterns: N bits ⇒ 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!



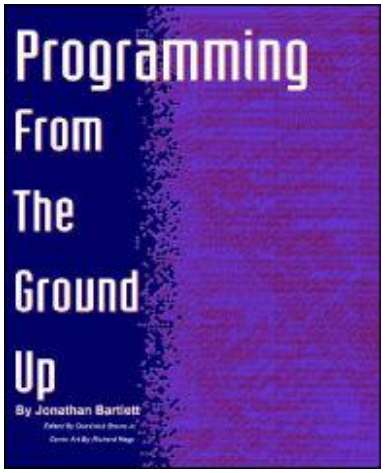
- Overflow: numbers ∞ ; computers finite,errors!

META: Ain't no free lunch

Assembly Language

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

Book: *Programming From the Ground Up*



"A new book was just released which is based on a new concept - teaching computer science through assembly language (Linux x86 assembly language, to be exact). This book teaches how the machine itself operates, rather than just the language. I've found that the key difference between mediocre and excellent programmers is whether or not they know assembly language. ***Those that do tend to understand computers themselves at a much deeper level.*** *Although [almost!] unheard of today, this concept isn't really all that new -- there used to not be much choice in years past. Apple computers came with only BASIC and assembly language, and there were books available on assembly language for kids. This is why the old-timers are often viewed as 'wizards': they* ***had*** *to know assembly language programming."*

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

Instruction Set Architectures

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

MIPS Architecture

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

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

Assembly Variables: Registers (1/4)

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

Assembly Variables: Registers (2/4)

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

Assembly Variables: Registers (3/4)

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

```
$0, $1, $2, … $30, $31
```

Assembly Variables: Registers (4/4)

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

  `$16 - $23` → `$s0 - $s7`

  (correspond to C variables)

  `$8 - $15` → `$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      # temp = g + h
add $t1,$s3,$s4      # temp = 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` (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 `int`s, then `p++;` would be `addi $s0 $s0 1`

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

“And in Conclusion…”

- **In MIPS Assembly Language:**
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- **New Instructions:**

  `add, addi, sub`
- **New Registers:**

  C Variables: `$s0 - $s7`

  Temporary Variables: `$t0 - $t9`

  Zero: `$zero`

2.2 Data Transfer & Decisions

Lecturer
Yuanqing
Cheng

Computer Architecture
（计算机体系结构）

Lecture 4 – Introduction to MIPS Data Transfer & Decisions I

2020-09-11

Review

- In MIPS Assembly Language:
  - Registers replace variables
  - One Instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- New Instructions:

  `add, addi, sub`
- New Registers:

  C Variables: `$s0 - $s7`

  Temporary Variables: `$t0 - $t7`

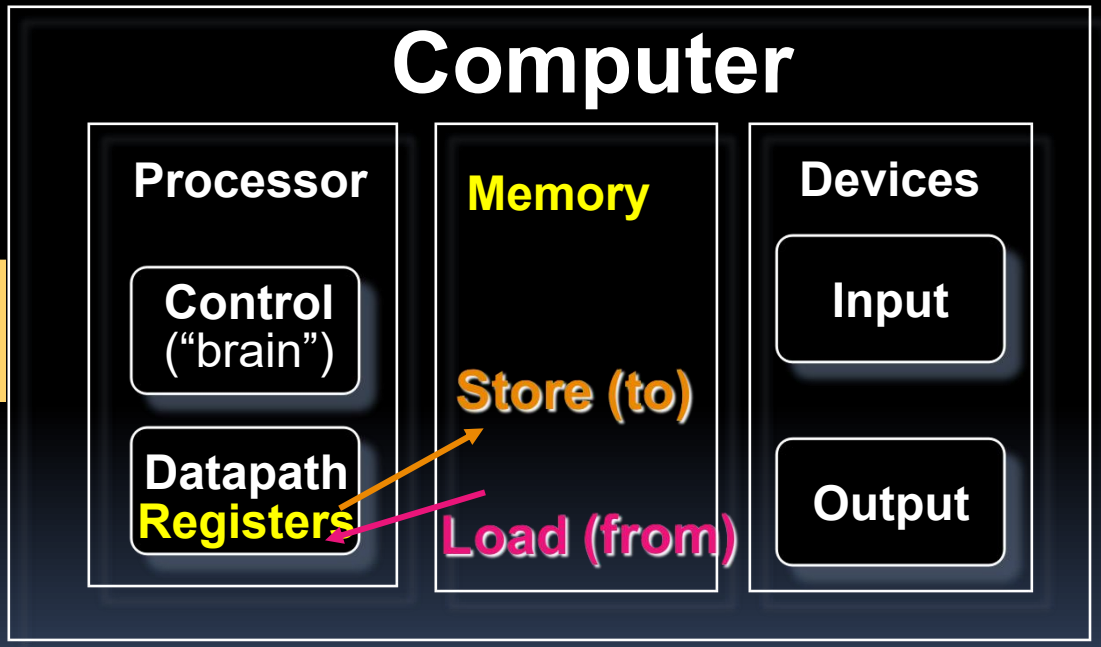
  Zero: `$zero`

Assembly Operands: Memory

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

Anatomy: 5 components of any Computer

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



These are "data transfer" instructions…

Data Transfer: Memory to Reg (1/4)

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

Data Transfer: Memory to Reg (2/4)

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

Data Transfer: Memory to Reg (3/4)

- **Load Instruction Syntax:**

```
1      2,3(4)
```

  - where
    1) operation name
    2) register that will receive value
    3) numerical offset in bytes
    4) register containing pointer to memory

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

Data Transfer: Memory to Reg (4/4)

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

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

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

Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load's
- MIPS Instruction Name:

  `sw` (meaning Store Word, so 32 bits or one word is stored at a time)

- 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], …`

- 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?
- 4x5=20 to select `A[5]`: byte v. word
- Compile by hand using registers:

```
g = h + A[5];
```

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

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

  - Add 20 to `$s3` to select `A[5]`, put into `$t0`
- Next add it to `h` and place in `g`

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

Notes about Memory

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

More Notes about Memory: Alignment

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

Last hex digit of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

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

Role of Registers vs. Memory

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

So Far...

- All instructions so far only manipulate data…we've built a calculator of sorts.
- In order to build a computer, we need ability to make decisions…
- C (and MIPS) provide labels to support "`goto`" jumps to places in code.
  - C: Horrible style; MIPS: Necessary!

- Heads up: pull out some papers and pens, you'll do an in-class exercise!

C Decisions: `if` Statements

- 2 kinds of `if` statements in C

  `if (`*condition*`)` *clause*

  `if (`*condition*`)` *clause1* `else` *clause2*

- Rearrange 2nd `if` into following:

```
if   (condition) goto L1;
     clause2;
     goto L2;
L1:  clause1;
L2:
```

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

MIPS Decision Instructions

- Decision instruction in MIPS:

```
beq   register1, register2, L1
```

`beq` is "Branch if (registers are) equal"

Same meaning as (using C):

```
if   (register1==register2) goto L1
```

- Complementary MIPS decision instruction

```
bne   register1, register2, L1
```

`bne` is "Branch if (registers are) not equal"

Same meaning as (using C):

```
if   (register1!=register2) goto L1
```

- Called conditional branches

MIPS Goto Instruction

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

```
j label
```

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

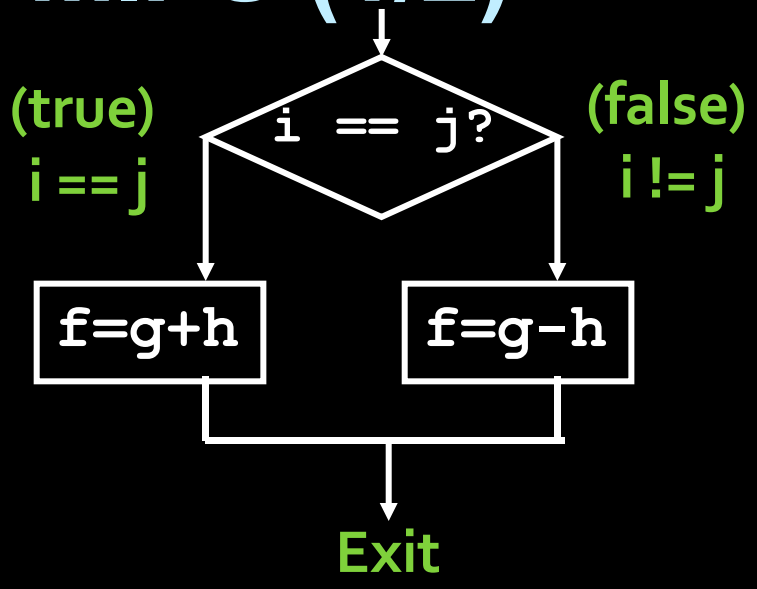
```
beq $0,$0,label
```

since it always satisfies the condition.

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

- Compile by hand

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



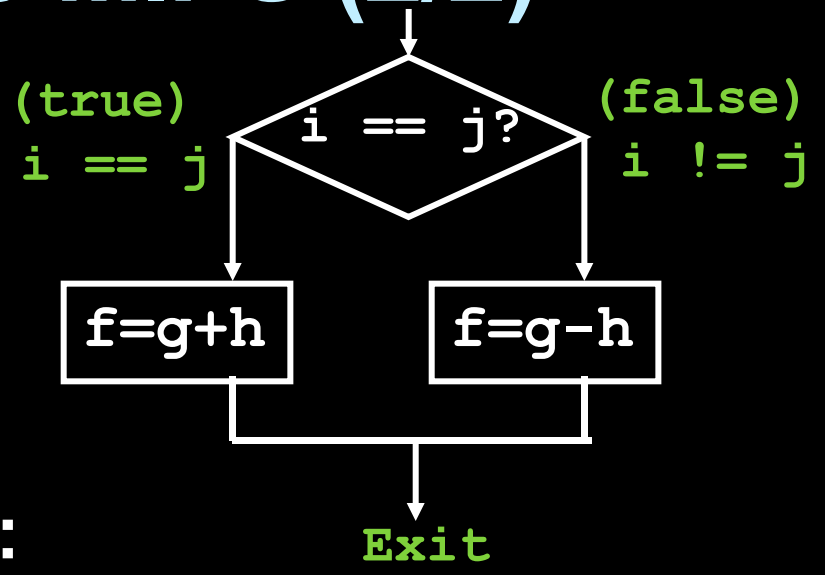
- Use this mapping:

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

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

- Compile by hand

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



- Final compiled MIPS code:

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

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

Peer Instruction

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

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

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

"And in Conclusion…"

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

  `lw, sw, beq, bne, j`

2.3 Addition to Data Transfer

Computer Architecture
（计算机体系结构）

Lecture 5
Introduction to MIPS : Decisions II

2020-09-14

Lecturer
Yuanqing
Cheng

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):

```
  15          1111
+  3        + 0011
----        ------
  18         10010
```

  - 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: $0000\ 0002_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
  - After: $0000\ 0008_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{two}$
  - 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 `int`s

```
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 >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement ≤ in one or more instructions using just `slt` and branches?
  - What about >?
  - What about ≥?

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
```

MIPS

```
Loop:   . . .

slti $t0,$s0,1      # $t0 = 1 if
                    # $s0<1 (g<1)
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`?

(`$s0 = FFFF FFFA`$_{hex}$, `$s1 = 0000 FFFA`$_{hex}$)

```
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)	j	>	2	&&	j	^	^	i
a)	j	>	2	&&	j	^	^	i
b)	j	>	2	&&	j	^	^	i
c)	j	>	2	&&	j	^	^	i
d)	j	>	2	||	j	^	^	i
e)	j	>	2	||	j	^	^	i
e)	j	>	2	||	j	^	^	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

Lecturer
Yuanqing
Cheng

Computer Architecture
（计算机体系结构）

Lecture 6 – Introduction to MIPS : Decisions II

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)

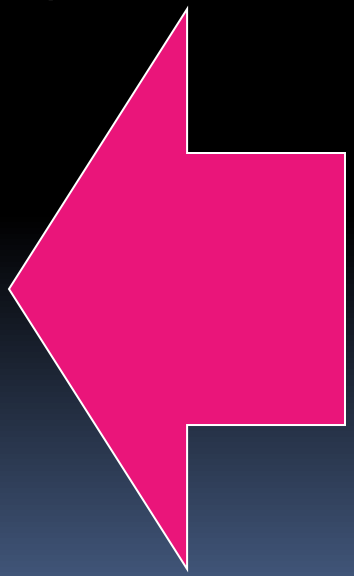
C

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

MIPS

address (shown in decimal)

```
1000
1004
1008
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)

C

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

MIPS

address (shown in decimal)

```
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)

C

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

MIPS

- 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.

```
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)

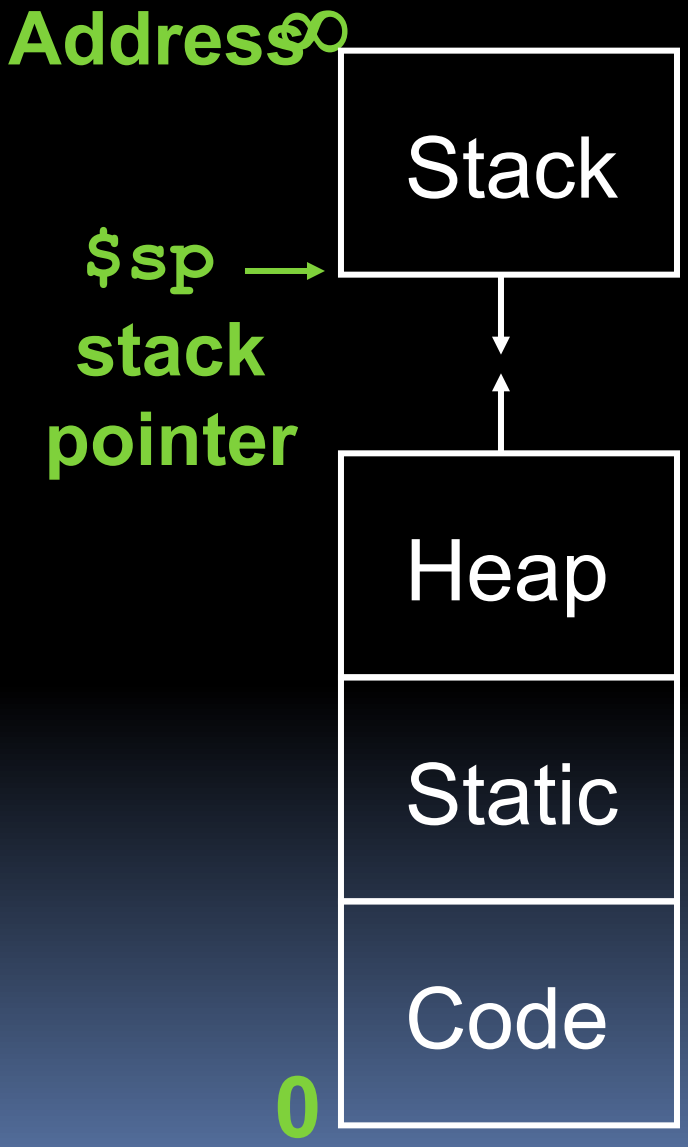
```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
```

- 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



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

```
sumSquare:
        addi $sp,$sp,-8  # space on stack
        sw $ra, 4($sp)   # save ret addr
"push"  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

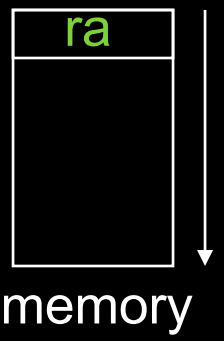
So what are they?

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	$16-$23	$s0-$s7
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…

	123
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

Lecturer
Yuanqing
Cheng

Computer Architecture
（计算机体系结构）

Lecture 7 – Introduction to MIPS Procedures II & Logical Ops

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 (`memory`)
  - Move items to saved rooms whose contents have already been moved to the garage (`memory`)
  - Optimize lifestyle (`code`) so that the amount you've got to move stuff back and forth from garage (`memory`) 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 (computation)
- What does friend (callee) do?
  - Friend takes what was in these rooms and puts them in the garage (memory)
  - 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 (memory) 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 & Shift Ops

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 **`and`**ing a bit with `0` produces a `0` at the output while `and`ing a bit with `1` produces the original bit.
- This can be used to create a mask.
  - Example:

```
      1011 0110 1010 0100 0011 1101 1001 1010
mask: 0000 0000 0000 0000 0000 1111 1111 1111
```

  - The result of **`and`**ing these:

```
      0000 0000 0000 0000 0000 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 `0`s).
- Thus, the **`and`** operator can be used to set certain portions of a bitstring to `0`s, 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   $t0,$t0,0xFFF
```

Uses for Logical Operators (3/3)

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

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

… **`$t0`** will contain **`0x1234FFFF`**

- (i.e., the high-order 16 bits are untouched, while the low-order 16 bits are forced to **`1`**s).

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…”

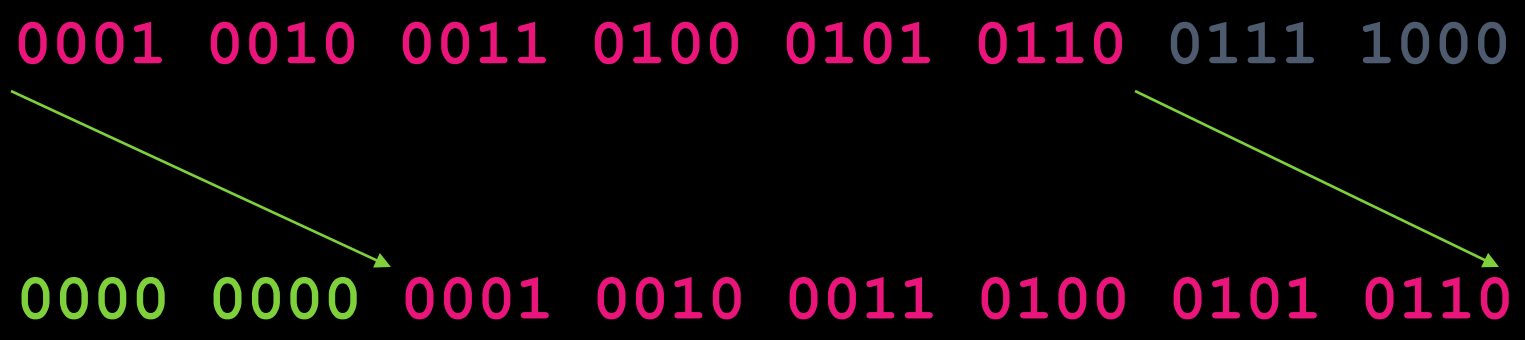
- Register Conventions: Each register has a purpose and limits to its usage. Learn these and follow them, even if you’re writing all the code yourself.
- Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either 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

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



  - Example: shift left by 8 bits

Shift Instructions (2/4)

- Shift Instruction Syntax:

  1 2,3,4

  …where

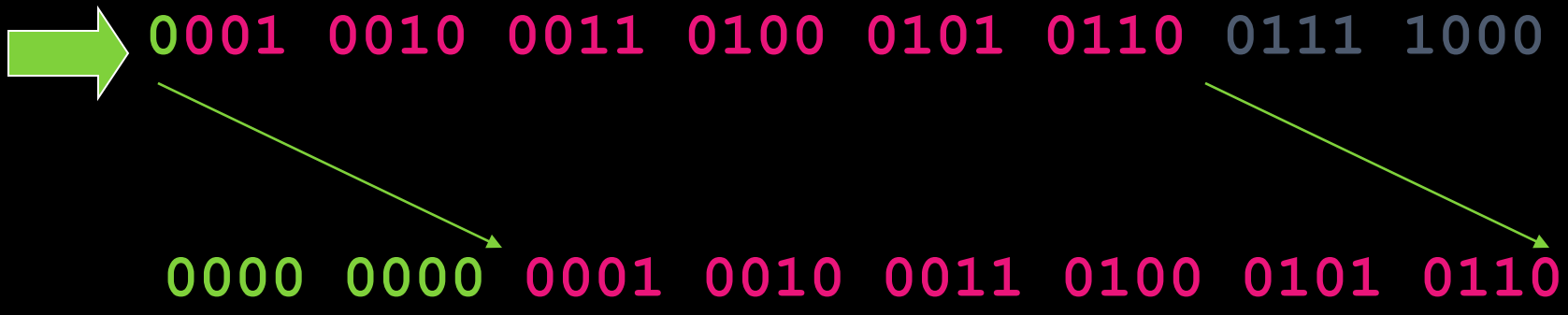
  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) shift amount (constant < 32)

- MIPS shift instructions:

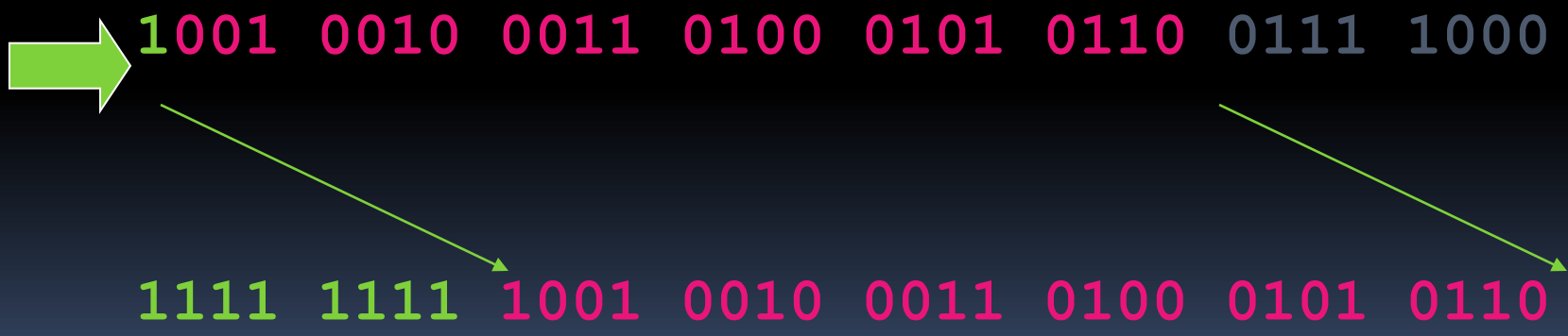
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

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 -∞)
  - 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
sw $a0, 0($sp)
jal fib
lw $a0, 0($sp)
addi $a0, $a0, -1
```

```
# $a0 = n - 1
# Need $a0 after jal
# fib(n - 1)
# restore $a0
# $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    # prod=0
      slt  $t1,$0,$a1     # mlr > 0?
      beq  $t1,$0,Fin     # no=>Fin
      add  $t0,$t0,$a0    # prod+=mc
      addi $a1,$a1,-1     # mlr-=1
      j    Loop           # goto Loop
Fin:
      add  $v0,$t0,$0     # $v0=prod
      jr   $ra            # return
```

```
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

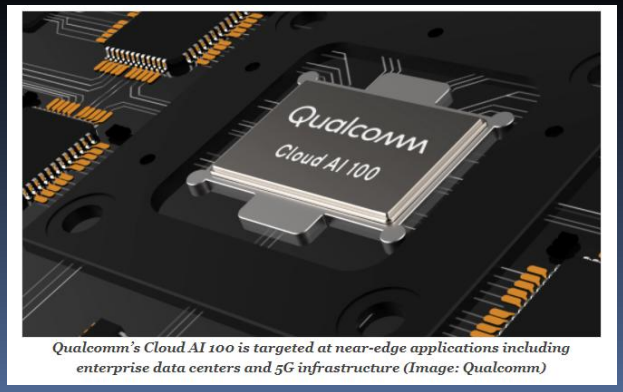
Computer Architecture
（计算机体系结构）

Lecture 8
MIPS Instruction Representation I

2020-09-18

Lecturer
Yuanqing
Ch

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)

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

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

Compiler

Assembly Language Program (e.g.,MIPS)

```
lw   $t0, 0($s2)
lw   $t1, 4($s2)
sw   $t1, 0($s2)
sw   $t0, 4($s2)
```

Assembler

Machine Language Program (MIPS)

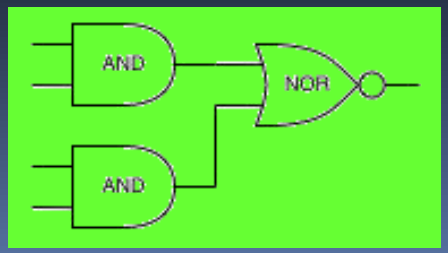
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



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: $012A\ 4020_{hex}$

  decimal representation: $19,546,144_{ten}$

  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

6	5	5	16

  - Again, each field has a name:

opcode	rs	rt	immediate

  - 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 FFCE$_{hex}$

decimal representation: 584,449,998$_{ten}$

Peer Instruction

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

a) add $0, $0, $0

b) subu $s0,$s0,$s0

c) lw $0, 0($0)

d) addi $0, $0, 35

e) subu $0, $0, $0

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_{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

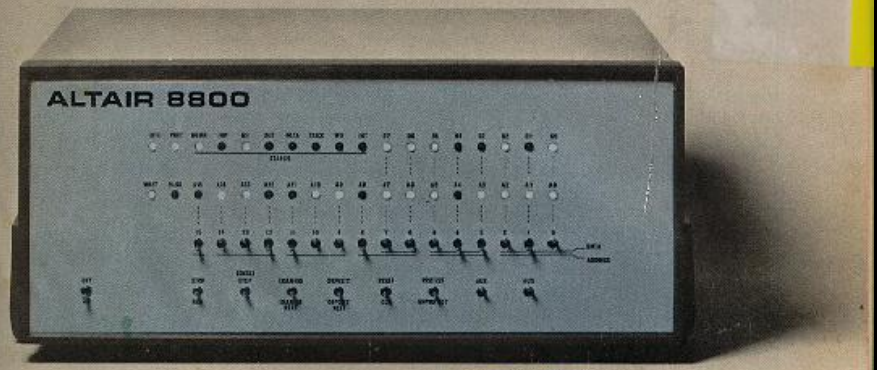
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		

Popular Electronics
JANUARY, 1975

PE TESTED

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 μs. 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 μs (min.)
Inputs and outputs: 256 (all directly addressable)
Number of basic machine instructions: 78 (181 with variants)
Add/substract time: 2 μs
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

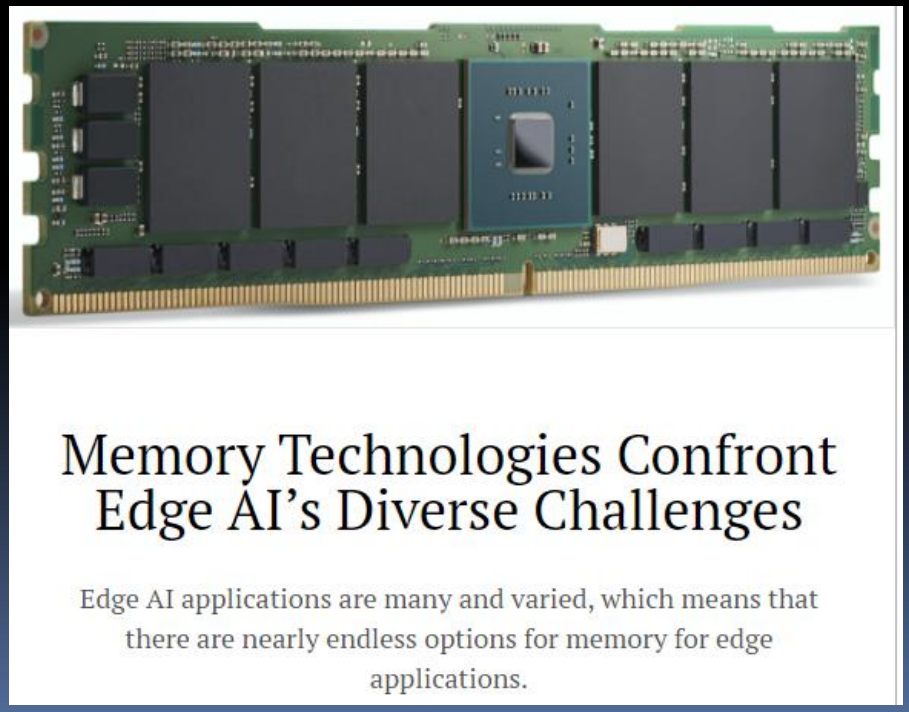
2.7 Formats of Instructions

Lecturer
Yuanqing
Cheng

Computer Architecture
（计算机体系结构）

Lecture 9
MIPS Instruction Representation II

2020-09-21



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 $2^{15} \le n < 2^{16}$ (I.e., with a `1` in the 15th bit and `0`s in the upper 2 bytes) as it does for numbers that are too large. $\Longrightarrow$

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 `0`s

I-Format Problems (3/3)

- Solution to Problem (continued):
  - So how does `lui` help us?
  - Example:

```
addiu $t0,$t0, 0xABABCDCD
```

…becomes

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

  - Now each I-format instruction has only a 16-bit immediate.
  - Wouldn't it be nice if the assembler would 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 = byte address of next instruction
  - If we do take the branch:

    PC = (PC + 4) + (`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, 11111111111111111111111111, 00 } = 10101111111111111111111111111100
  - 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)