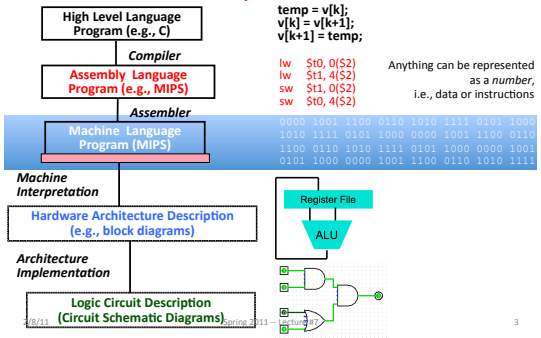
CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructions as Numbers

Instructors:
Randy H. Katz
David A. Patterson

http://inst.eecs.Berkeley.edu/~cs61c/fa10

Levels of Representation/ Interpretation



Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Key Concepts

- Inside computers, everything is a number
- But everything is of a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating point operations can lead to results too big to store within their representations: *overflow/underflow*

Number Representation

- Value of i-th digit is $d \times Base^i$ where i starts at 0 and increases from right to left:
- $123_{10} = 1_{10} \times 10_{10}^2 + 2_{10} \times 10_{10}^1 + 3_{10} \times 10_{10}^0$
  $= 1 \times 100_{10} + 2 \times 10_{10} + 3 \times 1_{10}$
  $= 100_{10} + 20_{10} + 3_{10}$
  $= 123_{10}$
- Binary (Base 2), Hexadecimal (Base 16), Decimal (Base 10) different ways to represent an integer
  - We use 1_{two}, 5_{ten}, 10_{hex} to be clearer (vs. 1_2, 4_8, 5_{10}, 10_{16})

Number Representation

- Hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- $FFF_{hex} = 15_{ten} \times 16_{ten}^2 + 15_{ten} \times 16_{ten}^1 + 15_{ten} \times 16_{ten}^0$
  $= 3840_{ten} + 240_{ten} + 15_{ten}$
  $= 4095_{ten}$
- $1111\ 1111\ 1111_{two} = FFF_{hex} = 4095_{ten}$
- May put blanks every group of binary, octal, or hexadecimal digits to make it easier to parse, like commas in decimal

Signed and Unsigned Integers

- C, C++, and Java have *signed integers*, e.g., 7, -255:

```
int x, y, z;
```

- C, C++ also have *unsigned integers*, which are used for addresses
- 32-bit word can represent 2^{32} binary numbers
- Unsigned integers in 32 bit word represent 0 to 2^{32}-1 (4,294,967,295)

Unsigned Integers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
… …
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 2{,}147{,}483{,}649_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2{,}147{,}483{,}650_{ten}$
… …
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

Signed Integers and Two's Complement Representation

- Signed integers in C; want ½ numbers <0, want ½ numbers >0, and want one 0
- *Two's complement* treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to 2^{31}-1 (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
  - Every computers uses two's complement today
- *Most significant bit* (leftmost) is the *sign bit*, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Two's Complement Integers

Sign Bit

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

Peer Instruction Question

- Suppose we had a 5 bit word. What integers can be represented in two's complement?

  Red: -32 to +31
  Orange: -31 to +32
  Green: 0 to +31
  Yellow: -16 to +15
  Pink: -15 to +15
  Blue: -15 to +16

Peer Instruction Answer

- Suppose we had a 5 bit word. What integers can be represented in two's complement?

  Red: -32 to +31
  Orange: -31 to +32
  Green: 0 to +31
  Yellow: -16 to +15
  Pink: -15 to +15
  Blue: -15 to +16

MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	`and`
Bit-by-bit OR	|	|	`or`
Bit-by-bit NOT	~	~	`nor`
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`

Bit-by-bit Definition

Operation	Input	Input	Output
AND	0	0	0
AND	0	1	0
AND	1	0	0
AND	1	1	1
OR	0	0	0
OR	0	1	1
OR	1	0	1
OR	1	1	1
NOR	0	0	1
NOR	0	1	0
NOR	1	0	0
NOR	1	1	0

Examples

- If register $t2 contains and
  $0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 1100\ 0000_{two}$
- Register $t1 contains
  $0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{two}$
- What is value of $t0 after:
  and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2

Student Roulette?

Examples

- If register $t2 contains and
  $0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 1100\ 0000_{two}$
- Register $t1 contains
  $0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{two}$
- What is value of $t0 after:
  and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2
  $0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{two}$

Examples

- If register $t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2

Student Roulette?

Examples

- If register $t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
  0000 0000 0000 0000 0011 1101 1100 0000_{two}

Examples

- If register $t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  nor $t0,$t1,$zero # reg $t0 = ~ (reg $t1 | 0)

Student Roulette?

Examples

- If register $t2 contains and
  0000 0000 0000 0000 0000 1101 1100 0000_{two}
- Register $t1 contains
  0000 0000 0000 0000 0011 1100 0000 0000_{two}
- What is value of $t0 after:
  nor $t0,$t1,$zero # reg $t0 = ~ (reg $t1 | 0)
  1111 1111 1111 1111 1100 0011 1111 1111_{two}

Shifting

- Shift left logical moves n bits to the left (insert 0s into empty bits)
  - Same as multiplying by 2^n for two's complement number
- For example, if register $s0 contained
  0000 0000 0000 0000 0000 0000 0000 $1001_{two} = 9_{ten}$
- If executed sll $s0, $s0, 4, result is:
  0000 0000 0000 0000 0000 0000 1001 $0000_{two} = 144_{ten}$
- And $9_{ten} \times 2_{ten}^4 = 9_{ten} \times 16_{ten} = 144_{ten}$
- Shift right logical moves n bits to the right (insert 0s into empty bits)
  - NOT same as dividing by 2n (negative numbers fail)

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained
  0000 0000 0000 0000 0000 0000 0001 $1001_{two} = 25_{ten}$
- If executed sra $s0, $s0, 4, result is:

Student Roulette?

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained
  0000 0000 0000 0000 0000 0000 0001 1001_{two}= 25_{ten}
- If executed sra $s0, $s0, 4, result is:
  0000 0000 0000 0000 0000 0000 0000 0001_{two}= 1_{ten}

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained
  1111 1111 1111 1111 1111 1111 1110 0111_{two}= -25_{ten}
- If executed sra $s0, $s0, 4, result is:

Student Roulette?

Shifting

- Shift right arithmetic moves n bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained
  1111 1111 1111 1111 1111 1111 1110 0111_{two}= -25_{ten}
- If executed sra $s0, $s0, 4, result is:
  1111 1111 1111 1111 1111 1111 1111 1110_{two}= -2_{ten}
- Unfortunately, this is NOT same as dividing by 2n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Impact of Signed and Unsigned Integers on Instruction Sets

- What (if any) instructions affected?
  - Load word, store word?
  - branch equal, branch not equal?
  - and, or, sll, srl?
  - add, sub, mult, div?
  - slti (set less than immediate)?

Student Roulette?

Peer Instruction Question

- C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't. Why?

Red:
  Logical operations AND and OR implement & and | while conditional branches implement && and ||

Orange:
  The previous statement has it backwards: && and || correspond to logical operations while & and | map to conditional branches

Green:
  They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C

Peer Instruction Answer

- C provides two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||) while MIPS doesn't. Why?

Red:
  Logical operations AND and OR implement & and | while conditional branches implement && and ||

Reason:
  e.g., && is logical `and`: true && true is true and everything else is false
  & is bitwise `and`: e.g., (1010_{two} & 1000_{two}) = 1000_{two}

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Administrivia

- Lab #4 posted (?)
- Project #1 Due Sunday @ 11:59:59
- Good news! No HW this week!
- Want in from the Wait List?
  - Sign up for Lab 019 (Friday, 7-9 PM – there is space available!)
- Midterm is now on the horizon:
  - No discussion during exam week
  - TA Review: Su, Mar 6, 2-5 PM, 2050 VLSB
  - Exam: Tu, Mar 8, 6-9 PM, 145/155 Dwinelle
  - Small number of special consideration cases, due to class conflicts, etc.—contact Dave or Randy

CS61c in the News

PDP = Peripheral Data Processor

Ken Olsen, Who Built DEC Into a Power, Dies at 84

By GLENN RIFKIN
Published: February 7, 2011

Ken Olsen, who helped reshape the computer industry as a founder of the Digital Equipment Corporation, at one time the world's second-largest computer company, died on Sunday. He was 84.

RECOMMEND
TWITTER
E-MAIL
PRINT
REPRINTS
SHARE

Enlarge This Image

Chitose Suzuki/Associated Press

Ken Olsen, the pioneering founder of DEC, in 1996.

His family announced the death but declined to provide further details. He had recently lived with a daughter in Indiana and had been a longtime resident of Lincoln, Mass.

Mr. Olsen, who was proclaimed "America's most successful entrepreneur" by Fortune magazine in 1986, built Digital on $70,000 in seed money, founding it with a partner in 1957 in the small Boston suburb of Maynard, Mass. With Mr. Olsen as its chief executive, it grew to employ more than 120,000 people at operations in more than 95 countries, surpassed in size only by I.B.M.

At its peak, in the late 1980s, Digital had $14 billion in sales and ranked among the most profitable companies in the nation.

But its fortunes soon declined after Digital began missing out on some critical market shifts, particularly toward the personal computer. Mr. Olsen was criticized as autocratic and resistant to new trends. "The personal computer will fall flat on its face in business,"

Don't Go Driving with Randy!

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Goals for Floating Point

- Standard arithmetic for reals for all computers
  - Like two's complement
- Keep as much precision as possible in formats
- Help programmer with errors in real arithmetic
  - +∞, -∞, Not-A-Number (NaN), exponent overflow, exponent underflow
- Keep encoding that is somewhat compatible with two's complement
  - E.g., 0 in Fl. Pt. is 0 in two's complement
  - Make it possible to sort without needing to do floating point comparison

Scientific Notation (e.g., Base 10)

- Normalized *scientific notation* (aka *standard form* or *exponential notation*):
  - r x E^i, *E* is exponent (usually 10), *i* is a positive or negative integer, *r* is a real number ≥ 1.0, < 10
  - Normalized => No leading 0s
  - 61 is 6.10 x 10^2, 0.000061 is 6.10 x10^{-5}
- r x E^i, *E* where is exponent, *i* is a positive or negative integer, *r* is a real number ≥ 1.0, < 10

Scientific Notation (e.g., Base 10)

- $(r \times e^i) \times (s \times e^j) = (r \times s) \times e^{i+j}$
  $(1.999 \times 10^2) \times (5.5 \times 10^3) = (1.999 \times 5.5) \times 10^5$
  $= 10.9945 \times 10^5$
  $= 1.09945 \times 10^6$
- $(r \times e^i) / (s \times e^j) = (r / s) \times e^{i-j}$
  $(1.999 \times 10^2) / (5.5 \times 10^3) = 0.3634545\ldots \times 10^{-1}$
  $= 3.634545\ldots \times 10^{-2}$
- For addition/subtraction, you first must align:
  $(1.999 \times 10^2) + (5.5 \times 10^3)$
  $= (.1999 \times 10^3) + (5.5 \times 10^3) = 5.6999 \times 10^3$

Which is Smaller? (i.e., closer to -∞)

- 0 vs. 1 x 10^{-127}?
- 1 x 10^{-126} vs. 1 x 10^{-127}?
- -1 x 10^{-127} vs. 0?
- -1 x 10^{-126} vs. -1 x 10^{-127}?

Student Roulette?

Which is Smaller? (i.e., closer to -∞)

- 0 vs. 1 x 10^{-127}?
- 1 x 10^{-126} vs. 1 x 10^{-127}?
- -1 x 10^{-127} vs. 0?
- -1 x 10^{-126} vs. -1 x 10^{-127}?

Floating Point: Representing Very Small Numbers

- Zero: Bit pattern of all 0s is encoding for 0.000
  - ⇒ But 0 in exponent should mean most negative exponent (want 0 to be next to smallest real)
  - ⇒ Can't use two's complement (1000 0000_{two})
- *Bias notation*: subtract bias from exponent
  - Single precision uses bias of 127; DP uses 1023
- 0 uses 0000 0000_{two} => 0-127 = -127;
  ∞, NaN uses 1111 1111_{two} => 255-127 = +128
  - Smallest SP real can represent: 1.00…00 x 2^{-126}
  - Largest SP real can represent: 1.11…11 x 2^{+127}

Bias Notation (+127)

How it is interpreted | How it is encoded

	Decimal Exponent	signed 2's complement	Biased Notation	Decimal Value of Biased Notation
∞, NaN	For infinities		11111111	255
	127	01111111	11111110	254
	...	...	...	...
	2	00000010	10000001	129
Getting closer to zero	1	00000001	10000000	128
	0	00000000	01111111	127
	-1	11111111	01111110	126
	-2	11111110	01111101	125
	...	...	...	...
	-126	10000010	00000001	1
Zero	For Denorms	10000001	00000000	0

What If Operation Result Doesn't Fit in 32 Bits?

- *Overflow*: calculate too big a number to represent within a word
- Unsigned numbers: 1 + 4,294,967,295 (2^{32}-1)
- Signed numbers: 1 + 2,147,483,647 (2^{31}-1)

Depends on the Programming Language

- C unsigned number arithmetic ignores overflow (arithmetic modulo 2^{32})

  1 + 4,294,967,295 =

Depends on the Programming Language

- C signed number arithmetic also ignores overflow

  1 + 2,147,483,647 (2^{31}-1) =

Depends on the Programming Language

- Other languages want overflow signal on signed numbers (e.g., Fortran)
- What's a computer architect to do?

MIPS Solution: Offer Both

- Instructions that can trigger overflow:
  - `add`, `sub`, `mult`, `div`, `addi`, `multi`, `divi`
- Instructions that don't overflow are called "unsigned" (really means "no overflow"):
  - `addu`, `subu`, `multu`, `divu`, `addiu`, `multiu`, `diviu`
- Given semantics of C, always use unsigned versions
- Note: `slt` and `slti` do signed comparisons, while `sltu` and `sltiu` do unsigned comparisons
  - Nothing to do with overflow
  - When would get different answer for `slt` vs. `sltu`?

What About *Real* Numbers in Base 2?

- r x E^i, *E* where is exponent (2), *i* is a positive or negative integer, *r* is a real number ≥ 1.0, < 2
- Computers version of normalized scientific notation called *Floating Point* notation

Floating Point Numbers

- 32-bit word has 2^{32} patterns, so must be approximation of real numbers ≥ 1.0, < 2
- IEEE 754 Floating Point Standard:
  - 1 bit for *sign (s)* of floating point number
  - 8 bits for *exponent (E)*
  - 23 bits for *fraction (F)*
    (get 1 extra bit of precision if leading 1 is implicit)

$(-1)^s \times (1 + F) \times 2^E$

- Can represent from 2.0×10^{-38} to 2.0×10^{38}

Floating Point Numbers

- What about bigger or smaller numbers?
- IEEE 754 Floating Point Standard:
  Double Precision (64 bits)
  - 1 bit for *sign (s)* of floating point number
  - 11 bits for *exponent (E)*
  - 52 bits for *fraction (F)*
    (get 1 extra bit of precision if leading 1 is implicit)

$(-1)^s \times (1 + F) \times 2^E$

- Can represent from 2.0×10^{-308} to 2.0×10^{308}
- 32 bit format called *Single Precision*

More Floating Point

- What about 0?
  - Bit pattern all 0s means 0, so no implicit leading 1
- What if divide 1 by 0?
  - Can get infinity symbols +∞, -∞
  - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? (∞ – ∞, 0 ÷ 0)
  - Can get special symbols NaN for Not-a-Number
  - Sign bit 0 or 1, largest exponent, not zero in fraction
- What if result is too big? ($2 \times 10^{308} \times 2 \times 10^{2}$)
  - Get *overflow* in exponent, alert programmer!
- What if result is too small? ($2 \times 10^{-308} \div 2 \times 10^{2}$)
  - Get *underflow* in exponent, alert programmer!

MIPS Floating Point Instructions

- C, Java has single precision (`float`) and double precision (`double`) types
- MIPS instructions: `.s` for single, `.d` for double
  - Fl. Pt. Addition single precision:
    Fl. Pt. Addition double precision:
  - Fl. Pt. Subtraction single precision:
    Fl. Pt. Subtraction double precision:
  - Fl. Pt. Multiplication single precision:
    Fl. Pt. Multiplication double precision:
  - Fl. Pt. Divide single precision:
    Fl. Pt. Divide double precision:

Student Roulette?

MIPS Floating Point Instructions

- C, Java has single precision (`float`) and double precision (`double`) types
- MIPS instructions: .s for single, .d for double
  - Fl. Pt. Addition single precision: add.s
    Fl. Pt. Addition double precision: add.d
  - Fl. Pt. Subtraction single precision: sub.s
    Fl. Pt. Subtraction double precision: sub.d
  - Fl. Pt. Multiplication single precision: mul.s
    Fl. Pt. Multiplication double precision: mul.d
  - Fl. Pt. Divide single precision: div.s
    Fl. Pt. Divide double precision: div.d

MIPS Floating Point Instructions

- C, Java have single precision (`float`) and double precision (`double`) types
- MIPS instructions: `.s` for single, `.d` for double
  - Fl. Pt. Comparison single precision:
    Fl. Pt. Comparison double precision:
  - Fl. Pt. branch:
- Since rarely mix integers and Floating Point, MIPS has separate registers for floating-point operations: `$f0`, `$f1`, …, `$f31`
  - Double precision uses adjacent even-odd pairs of registers:
  - `$f0` and `$f1`, `$f2` and `$f3`, `$f4` and `$f5`, …, `$f30` and `$f31`
- Need data transfer instructions for these new registers
  - `lwc1` (load word), `swc1` (store word)
  - Double precision uses two `lwc1` instructions, two `swc1` instructions

Peer Instruction Question

Suppose Big, Tiny, and BigNegative are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or 4.32×10^{17}), Tiny to a small number (e.g., seconds/ femtosecond or 1.0×10^{-15}), BigNegative = - Big.
Here are two conditionals:

I. (Big * Tiny) * BigNegative == (Big * BigNegative) * Tiny

II. (Big + Tiny) + BigNegative == (Big + BigNegative) + Tiny

Which statement about these is correct?

Red. I. is false and II. is false

Orange. I. is false and II. is true

Yellow. I. is true and II. is false

Green. I. is true and II. is true

Peer Instruction Answer

Suppose Big, Tiny, and BigNegative are floats in C, with Big initialized to a big number (e.g., age of universe in seconds or 4.32×10^{17}), Tiny to a small number (e.g., seconds/femtosecond or 1.0×10^{-15}), BigNegative = - Big.
Here are two conditionals:

I. (Big * Tiny) * BigNegative == (Big * BigNegative) * Tiny

II. (Big + Tiny) + BigNegative == (Big + BigNegative) + Tiny

Which statement about these is correct?

Yellow. I. is true and II. is false (if we don't consider overflow)—but there are cases where one side overflows while the other does not

I. Works ok if no overflow, but because exponents add, if Big * BigNeg overflows, then result is overflow, not -1

II. Left hand side is 0, right hand side is tiny

Pitfalls

- Floating point addition is NOT associative
- Some optimizations can change order of floating point computations, which can change results
- Need to ensure that floating point algorithm is correct even with optimizations

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Key Concepts

- MIPS ISA guided by four RISC design principles:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

Instructions as Numbers

- Instructions are also kept as binary numbers in memory
  - Stored program concept
  - As easy to change programs as it is to change data
- Register names mapped to numbers
- Need to map instruction operation to a part of number

Instructions as Numbers

- `addu $t0,$s1,$s2`
  - Destination register `$t0` is register 8
  - Source register `$s1` is register 17
  - Source register `$s2` is register 18
  - Add unsigned instruction encoded as number 33

0	17	18	8	0	33
000000	10001	10010	01000	00000	100001
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Groups of bits call *fields* (unused field default is 0)
- Layout called *instruction format*
- Binary version called *machine instruction*

Instructions as Numbers

- `sll $zero,$zero,0`
  - `$zero` is register 0
  - Shift amount 0 is 0
  - Shift left logical instruction encoded as number 0

0	0	0	0	0	0
000000	00000	00000	00000	00000	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Can also represent machine code as base 16 or base 8 number: $0000\ 0000_{hex}$, 0000000000_{oct}

Everything in a Computer is Just a Binary Number

- Up to program to decide what data means
- Example 32-bit data shown as binary number:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$

What does it mean if its treated as

1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters
6. *MIPS instruction*

Student Roulette?

Implications of Everything is a Number

- *Stored program concept*
  - Invented about 1947 (many claim invention)
- As easy to change programs as to change data!
- Implications?

Student Roulette?

Names of MIPS fields

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *op*: Basic operation of instruction, or *opcode*
- *rs*: 1st register source operand
- *rt*: 2nd register source operand.
- *rd*: register destination operand (result of operation)
- *shamt*: Shift amount.
- *funct*: Function. This field, often called *function code*, selects the specific variant of the operation in the op field

What about Load, Store, Immediate, Branches, Jumps?

- Fields for constants only 5 bits (-16 to +15)
  - Too small for many common cases
- #1 Simplicity favors regularity (all instructions use one format) vs. #3 Make common case fast (multiple instruction formats)?
- 4th Design Principle: *Good design demands good compromises*
- Better to have multiple instruction formats and keep all MIPS instructions same *size*
  - All MIPS instructions are 32 bits or 4 bytes

Names of MIPS Fields in I-type

op	rs	rt	address or constant
6 bits	5 bits	5 bits	16 bits

- *op*: Basic operation of instruction, or *opcode*
- *rs*: 1st register source operand
- *rt*: 2nd register source operand for branches but register destination operand for lw, sw, and immediate operations
- *Address/constant*: 16-bit two's complement number
  - Note: equal in size of rd, shamt, funct fields

Register (R), Immediate (I), Jump (J) Instruction Formats

R-type	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I-type	op	rs	rt	address or constant		
	6 bits	5 bits	5 bits	16 bits		

- Now loads, stores, branches, and immediates can have 16-bit two's complement address or constant: -32,768 (-2^{15}) to +32,767 ($2^{15}-1$)
- What about jump, jump and link?

J-type	op	address
	6 bits	26 bits

Encoding of MIPS Instructions: Must Be Unique!

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	*n.a.*
subu	R	0	reg	reg	reg	0	35_{ten}	*n.a.*
sltu	R	0	reg	reg	reg	0	43_{ten}	*n.a.*
sll	R	0	reg	*n.a*	reg	constant	0_{ten}	*n.a.*
addi unsigned	I	9_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	constant
lw (load word)	I	35_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
sw (store word)	I	43_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
beq	I	4_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
bne	I	5_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
j (jump)	J	2_{ten}	*n.a.*	*n.a.*	*n.a.*	*n.a.*	*n.a.*	address
jal	J	3_{ten}	*n.a.*	*n.a.*	*n.a.*	*n.a.*	*n.a.*	address
jr (jump reg)	R	0	reg	reg	reg	0	8_{ten}	*n.a.*

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Agenda

- Everything is a Number
- Administrivia
- Overflow and Real Numbers
- Instructions as Numbers
- Technology Break
- Assembly Language to Machine Language
- Summary

Converting C to MIPS Machine code

&A=$t0 (reg 8), $t1 (reg 9), h=$s2 (reg 18)

A[300] = h + A[300];

Format?

lw $t0,1200($t1) _

addu $t1,$s2,$t0 _

sw $t0,1200($t1) _

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	*n.a.*
lw (load word)	I	35_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
sw (store word)	I	43_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

&A=$t0 (reg 8), $t1 (reg 9), h=$s2 (reg 18)

A[300] = h + A[300];

Format?

lw $t0,1200($t1) _	35	9	8	1200		
addu $t1,$s2,$t0 _	0	18	8	9	0	33
sw $t0,1200($t1) _	43	9	8	1200		

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	*n.a.*
lw (load word)	I	35_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address
sw (store word)	I	43_{ten}	reg	reg	*n.a.*	*n.a.*	*n.a.*	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Addressing in Branches

I-type	op	rs	rt	address or constant
	6 bits	5 bits	5 bits	16 bits

- Programs much bigger than 2^{16} bytes, but branch address must fit in 16-bit field
  - Must specify a register for branch addresses for big programs: PC = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: *PC-relative branching*

Addressing in Branches

I-type	op	rs	rt	address or constant
	6 bits	5 bits	5 bits	16 bits

- Hardware increments PC early, so relative address is PC = (PC + 4) + Branch address
- Another optimization since all MIPS instructions 4 bytes long?
- Multiply value in branch address field by 4!
- MIPS PC-relative branching
  PC = (PC + 4) + (Branch address*4)

Addressing in Jumps

J-type	op	address
	6 bits	26 bits

- Same trick for Jumps, Jump and Link
  PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

Converting to MIPS Machine code

Address	Loop:	Format?
800	sll $t1,$s3,2	_
804	addu $t1,$t1,$s6	_
808	lw $t0,0($t1)	_
812	bne $t0,$s5, Exit	_
816	addiu $s3,$s3,1	_
820	j Loop	_

Exit:

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting to MIPS Machine code

Address	Loop:	Format?						
800	sll $t1,$s3,2	R	0	0	19	9	2	0
804	addu $t1,$t1,$s6	R	0	9	22	9	0	33
808	lw $t0,0($t1)	I	35	9	8	0		
812	bne $t0,$s5, Exit	I	5	8	21	2		
816	addiu $s3,$s3,1	I	8	19	19	1		
820	j Loop	J	2	200				
	Exit:							

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

32 bit Constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- *Load Upper Immediate (*`lui` *or* "Louie") puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0? $0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_{two}$

lui $s0, 61 # 61 = $0000\ 0000\ 0011\ 1101_{two}$

ori $s0, $s0, 2304 # 2304 = $0000\ 1001\ 0000\ 0000_{two}$

Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called *assembly,* program called *assembler*
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called *assembly language*
- Can also accept instructions convenient for programmer but not in hardware
  - *Load immediate* (`li`) allows 32-bit constants, assembler turns into `lui` + `ori` (if needed)
  - *Load double* (`ld`) uses two lwc1 instructions to load a pair of 32-bit floating point registers
  - Called *Pseudo-Instructions*

"And in Conclusion, …"

- Program can interpret binary number as unsigned integer, two's complement signed integer, floating point number, ASCII characters, Unicode characters, …
- Integers have largest positive and largest negative numbers, but represent all in between
  - Two's comp. weirdness is one extra negative numInteger and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Floating point is an approximation of reals
- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- Assemblers can enhance machine instruction set to help assembly-language programmer