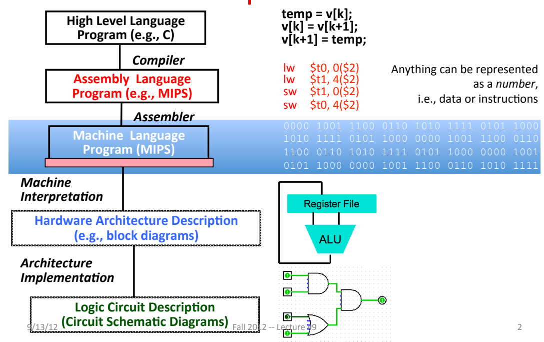
CS 61C:
Great Ideas in Computer Architecture
Instructions as Numbers
and *Floating Point Numbers*

Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

Big Idea #1: Levels of Representation/ Interpretation



Agenda

- Review
- Instructions as Numbers
- Administrivia
- Floating Point Numbers
- And in Conclusion, …

Optimized Function Convention

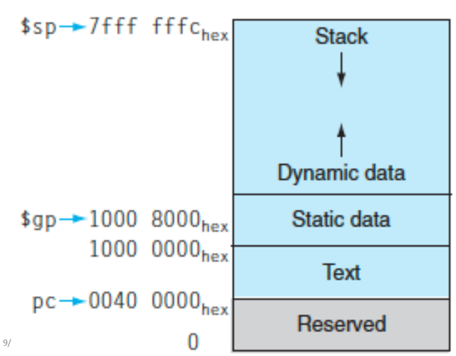
To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   – Caller can rely on values being unchanged
   – `$ra`, `$sp`, `$gp`, `$fp`, "saved registers" `$s0`- `$s7`
2. Not preserved across function call
   – Caller *cannot* rely on values being unchanged
   – Return value registers `$v0`,`$v1`, Argument registers `$a0`-`$a3`, "temporary registers" `$t0`-`$t9`

Where is the Stack in Memory?

- MIPS convention
- Stack starts in high memory and grows down
  – Hexadecimal (base 16) : 7fff $fffc_{hex}$
- MIPS programs (*text segment*) in low end
  – 0040 0000_{hex}
- *static data segment* (constants and other static variables) above text for static variables
  – MIPS convention *global pointer* (`$gp`) points to static
  – (30 of 32, 2 left! – will see when talk about OS)
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

MIPS Memory Allocation



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

Twos Complement Examples

- Assume for simplicity 4 bit width, -8 to +7 represented

```
   3   0011          3   0011          -3   1101
  +2   0010     + (-2)   1110     + (-2)    1110
   5   0101          1 1 0001          -5 1 1011

   7   0111         -8   1000
  +1   0001     + (-1)   1111
  -8   1000         +7 1 0111
```

Overflow! ***Underflow!***

Carry into MSB = Carry Out MSB

Carry into MSB ≠ Carry Out MSB

Agenda

- Review
- Instructions as Numbers
- Administrivia
- Floating Point Numbers
- And in Conclusion, …

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*

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?

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

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

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}

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.

Converting C to MIPS Machine code

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

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

Format?

lw $t0,1200($t1)	_		
addu $t0,$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

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

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

Format?

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

Agenda

- Review
- Floating Point Numbers
- Administrivia
- Instructions as Numbers
- And in Conclusion, …

CS61c in the News

- iPhone 5, 9/12/12

  - ARM Cortex A-15 core
  - Claimed 2x Performance of iPhone 4's A5 Chip (fabricated by Samsung!)
  - Still dual core

CS 61c in the News

- 1-1.5 Ghz, dual core
- Level 1 caches: 32KB instruction and 32KB data, with cache coherence
- "out-of-order superscalar pipeline with a tightly-coupled low-latency level-2 cache up to 4MB in size"
  - 15 stage integer / 17-25 stage floating point pipeline, with out-of-order speculative issue 3-way superscalar execution pipeline
- "full hardware virtualization, Large Physical Address Extensions (LPAE) addressing (40 bit) to 1TB, error correction capability for fault-tolerance and soft-fault recovery"
- "hw support for data management and arbitration, enabling multiple software environments and apps to simultaneously access the system capabilities"

Agenda

- Review
- Instructions as Numbers
- Administrivia
- Floating Point Numbers
- And in Conclusion, …

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}

Scientific Notation (e.g., Base 10)

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

Which is Less? (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}?

Which is Less? (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}

9/14/12 Fall 2012 -- Lecture #9 31

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
	. . .	. . .	. . .	. . .
Getting closer to zero	2	00000010	10000001	129
	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

9/14/12 Fall 2012 -- Lecture #9 32

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)

9/14/12 Fall 2012 -- Lecture #9 33

Depends on the Programming Language

- C unsigned number arithmetic ignores overflow (arithmetic modulo 2^{32})
  1 + 4,294,967,295 =

9/14/12 Fall 2012 -- Lecture #9 Student Roulette? 34

Depends on the Programming Language

- C unsigned number arithmetic ignores overflow (arithmetic modulo 2^{32})
  1 + 4,294,967,295 = $FFFF_{hex}$ + 1 = 0

9/14/12 Fall 2012 -- Lecture #9 35

Depends on the Programming Language

- C signed number arithmetic also ignores overflow
  1 + 2,147,483,647 (2^{31}-1) =

9/14/12 Fall 2012 -- Lecture #9 Student Roulette? 36

Depends on the Programming Language

- C signed number arithmetic also ignores overflow
  1 + 2,147,483,647 (2^{31}-1) = 1 + $EFFF_{hex}$ = $FFFF_{hex}$ = -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`?

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`?
  - -1 < 0 signed, but $FFFF_{hex}$ > 0 unsigned!

What About *Real* Numbers in Base 2?

- r x E^i, *E* where exponent is (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 x 10^{-38} to 2.0 x 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? ($2\text{x}10^{308} \times 2\text{x}10^{2}$)
  - Get *overflow* in exponent, alert programmer!
- What if result is too small? ($2\text{x}10^{-308} \div 2\text{x}10^{2}$)
  - Get *underflow* in exponent, alert programmer!

Floating Point Add Associativity?

- A = (1000000.0 + 0.000001) - 1000000.0
- B = (1000000.0 - 1000000.0) + 0.000001
- In single precision floating point arithmetic, A does not equal B

  A = 0.000000, B = 0.000001
- Floating Point Addition is not Associative!
  - Integer addition is associative
- When does this matter?

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:

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 x 10^{17}), Tiny to a small number (e.g., seconds/femtosecond or 1.0 x 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?

Orange. I. is false and II. is false

Green. I. is false and II. is true

Pink. I. is true and II. is false

Yellow. 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 x 10^{17}), Tiny to a small number (e.g., seconds/femtosecond or 1.0 x 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

"And in Conclusion, …"

- Program can interpret binary number as unsigned integer, two's complement signed integer, floating point number, ASCII characters, Unicode characters, … even instructions!
- 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