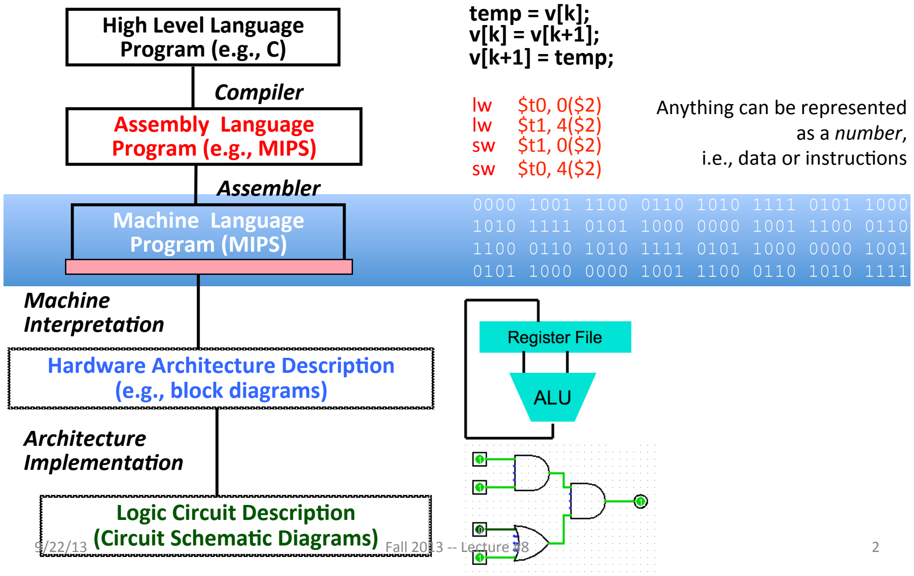
CS 61C: Great Ideas in Computer Architecture *Instructions as Numbers* and *Floating Point Numbers*

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa13

Big Idea #1: Levels of Representation/ Interpretation



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

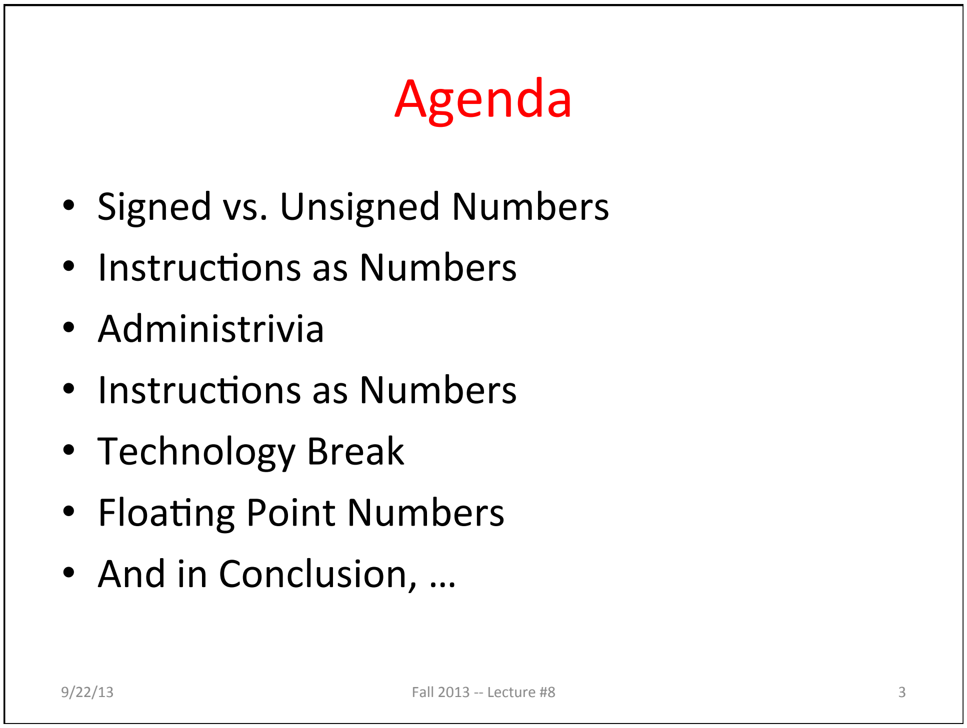
Logic Circuit Description (Circuit Schematic Diagrams)

```
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)
```

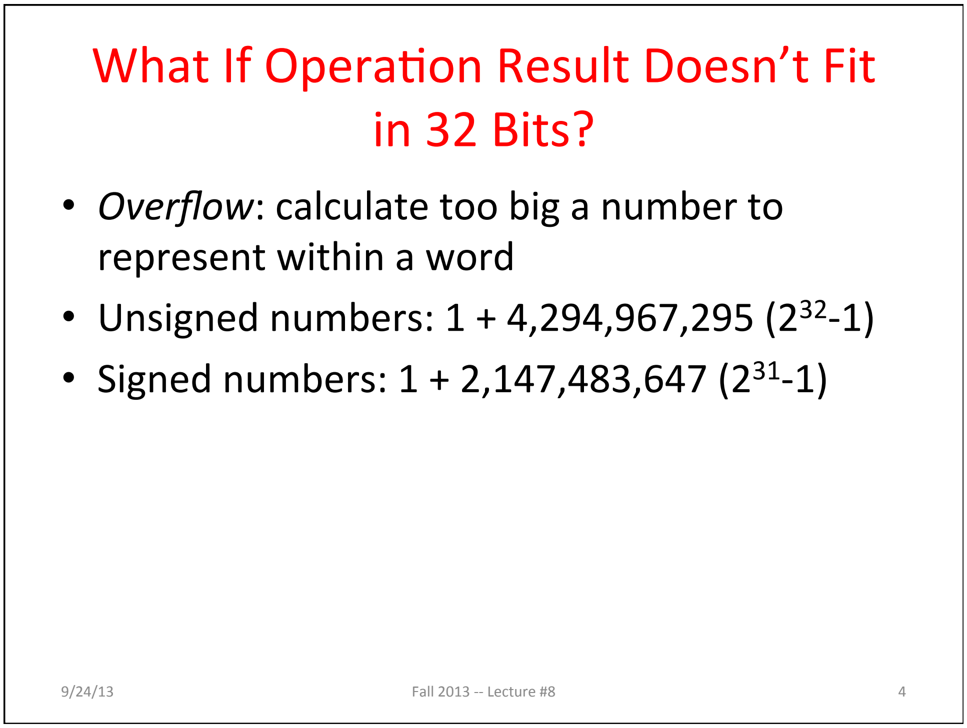
Anything can be represented as a *number*, i.e., data or instructions

```
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
```



Agenda

- Signed vs. Unsigned Numbers
- Instructions as Numbers
- Administrivia
- Instructions as Numbers
- Technology Break
- Floating Point Numbers
- And in Conclusion, …



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

  1 + 4,294,967,295 = $\mathrm{FFFFFFFF}_{hex}$ + 1 = 0

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

- C signed number arithmetic also ignores overflow

  1 + 2,147,483,647 (2^{31}-1) = 1 + $7FFFFFFF_{hex}$ = 80000000_{hex} = -2,147,483,648

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

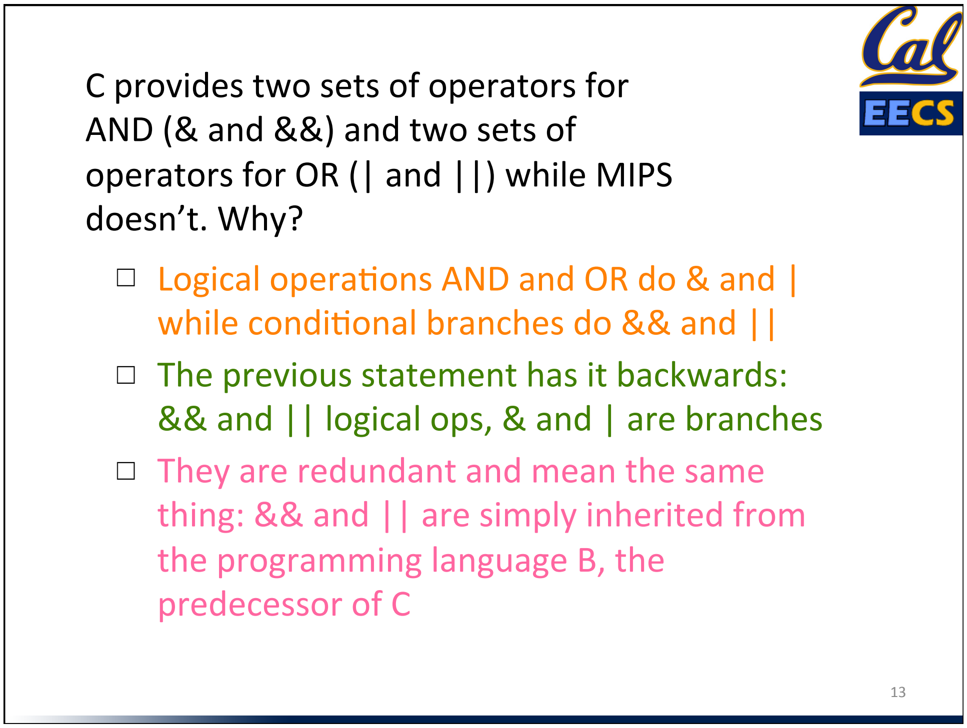
- Instructions that can trigger overflow:
  - `add`, `sub`, `mult`, `div`, `addi`, `multi`, `divi`
- Instructions that don't overflow are called "unsigned" (really means "no overflow"):
  - `addu`, `subu`, `multu`, `divu`, `addiu`, `multiu`, `diviu`
- Given semantics of C, always use unsigned versions
- 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!

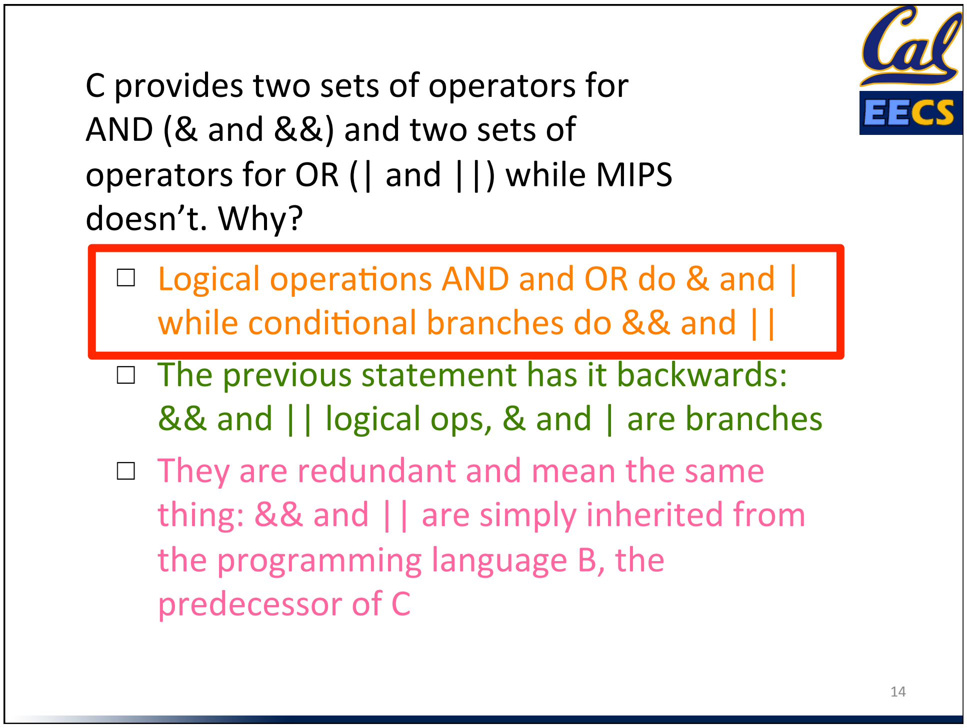
Impact of Signed and Unsigned Integers on Instruction Sets

- 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)?



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

- ☐ Logical operations AND and OR do & and | while conditional branches do && and ||
- ☐ The previous statement has it backwards: && and || logical ops, & and | are branches
- ☐ They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C



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

- ☐ Logical operations AND and OR do & and | while conditional branches do && and ||
- ☐ The previous statement has it backwards: && and || logical ops, & and | are branches
- ☐ They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C

Agenda

- Signed vs. Unsigned Numbers
- Instructions as Numbers
- Administrivia
- Instructions as Numbers Contintuted
- Technology Break
- 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?

RISC ISA Design Principles

#1: Simplicity Favors Regularity

Small number of formats, fixed fields

#2: Smaller is Faster

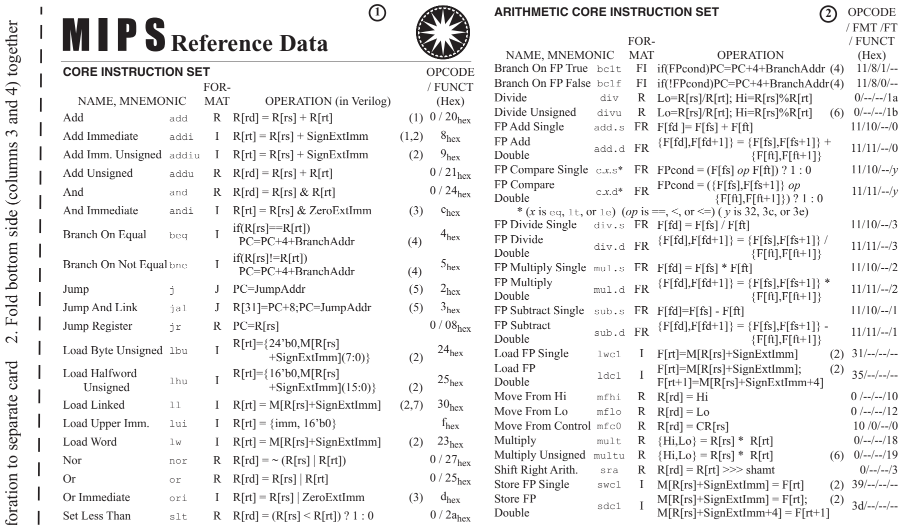
Few instructions, basic primitives

#3: Good Design Demands Good Compromises

Balance usefulness and simplicity

#4: *Make the Common Case Fast*

MIPS ISA



foration to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	`add`	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	`addiu`	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	`addu`	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	`and`	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	`andi`	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	`jal`	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	`jr`	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	`lbu`	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	`lhu`	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	`ll`	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	`nor`	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	`or`	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	`ori`	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	`slt`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	`bc1t`	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	`bc1f`	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	`div`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	`divu`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	`add.s`	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	`add.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	`c.x.s*`	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	`c.x.d*`	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	`div.s`	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	`div.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	`mul.s`	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	`mul.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	`sub.s`	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	`sub.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	`lwc1`	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	`ldc1`	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	`mfhi`	R	R[rd] = Hi		0 /--/--/10
Move From Lo	`mflo`	R	R[rd] = Lo		0 /--/--/12
Move From Control	`mfc0`	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	`mult`	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	`multu`	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	`sra`	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	`swc1`	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	`sdc1`	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=mips_reference_data.pdf

---

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}, 00000000000_{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 (justify 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

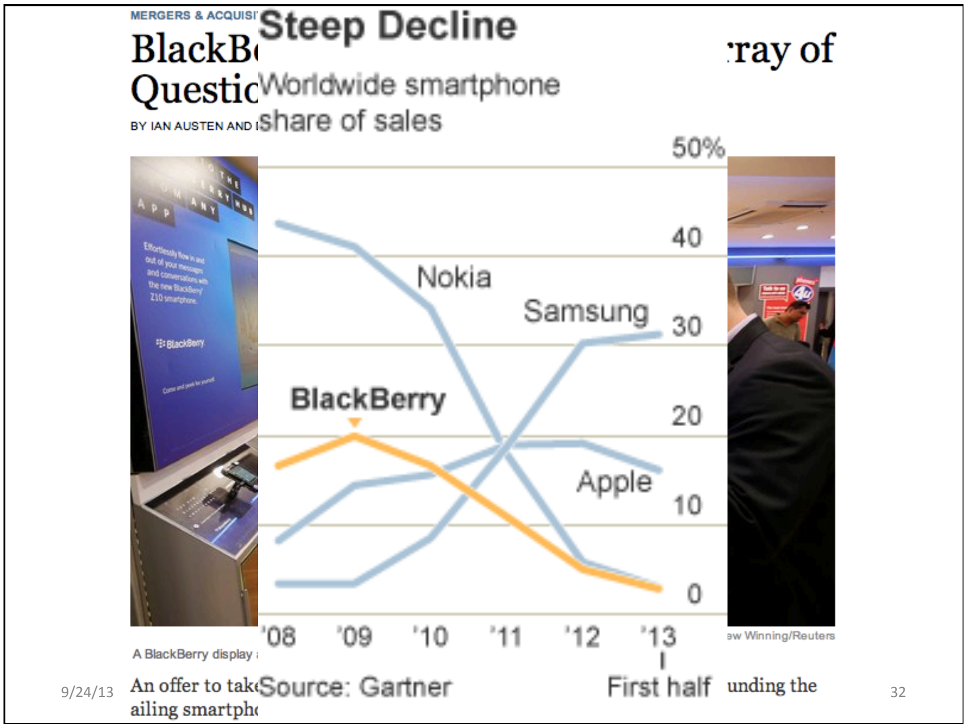
R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Agenda

- Signed vs. Unsigned Numbers
- Instructions as Numbers
- Administrivia
- Instructions as Numbers (continued)
- Technology Break
- Floating Point Numbers
- And in Conclusion, …

Administrivia

- Tuesday discussion sections henceforth cancelled:
  - Sagar: Tu 9-10 → Friday, 8-9, 405 Soda
  - Sung Roa: Tu 10-11 → Friday, 1-2, 320 Soda
- Piazza Etiquette
  - Please *don't* post your code
  - Please *don't* post copyrighted material
  - Remember *do* look before you post



MERGERS & ACQUISI

BlackB
Questic

:ray of

BY IAN AUSTEN AND I

Steep Decline

Worldwide smartphone share of sales

Source: Gartner

A BlackBerry display

ew Winning/Reuters

An offer to tak
ailing smartph

unding the

Agenda

- Signed vs. Unsigned Numbers
- Instructions as Numbers
- Administrivia
- **Instructions as Numbers (cont.)**
- Technology Break
- Floating Point Numbers
- And in Conclusion, …

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

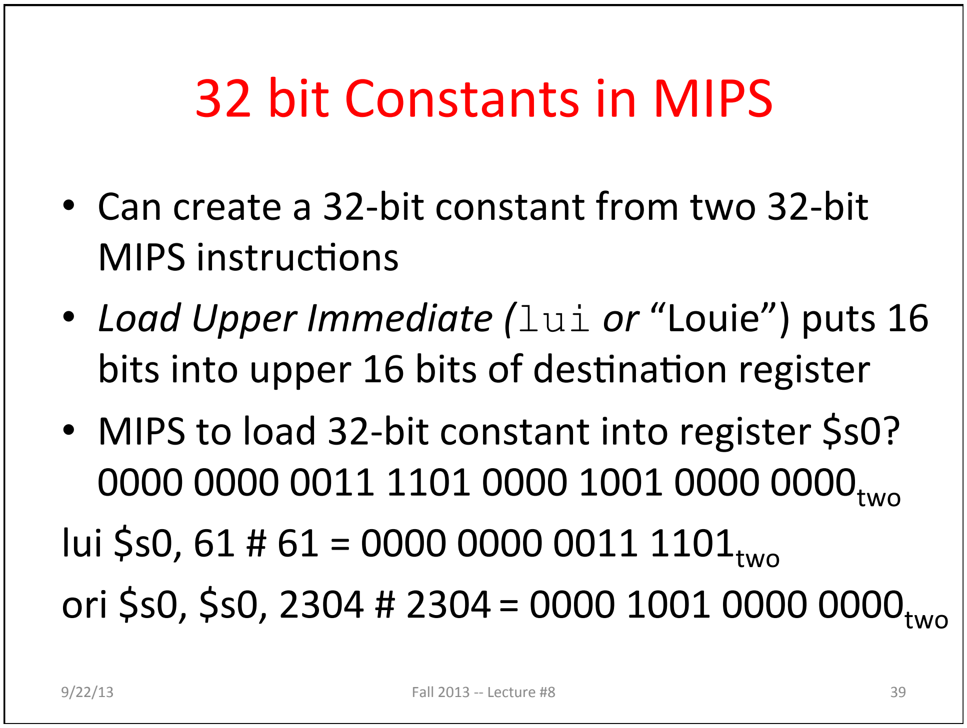
- Same trick for Jumps, Jump and Link
  PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

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				

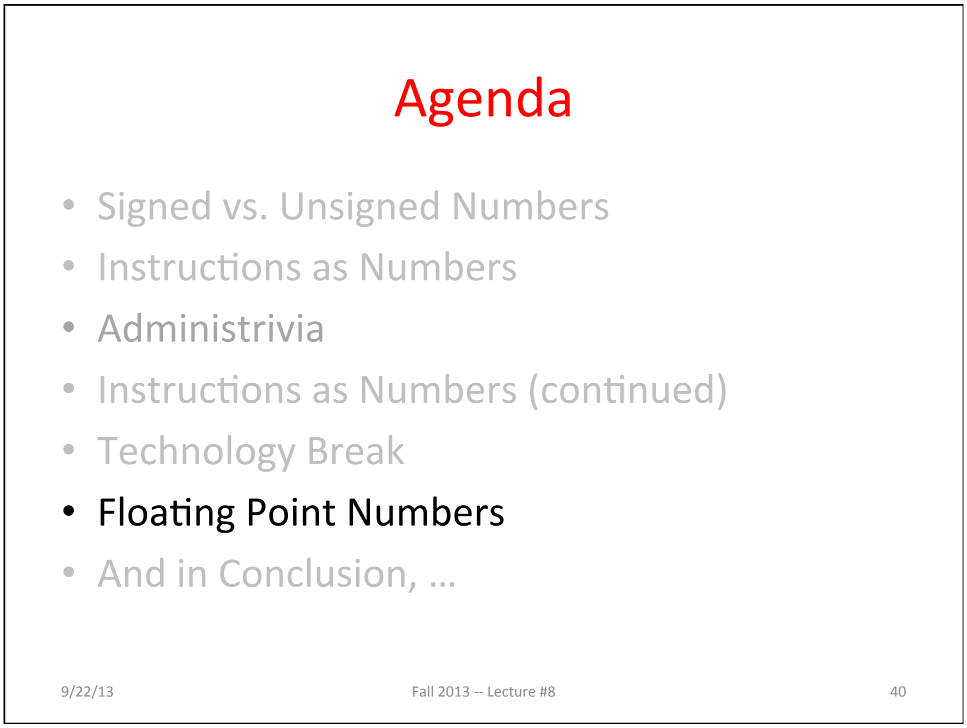


32 bit Constants in MIPS

- 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}$



Agenda

- Signed vs. Unsigned Numbers
- Instructions as Numbers
- Administrivia
- Instructions as Numbers (continued)
- Technology Break
- 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*):
  - $r \times E^i$, *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 6.10×10^2, 0.000061 is 6.10×10^{-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}?

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 ($1000\ 0000_{\text{two}}$)
- *Bias notation*: subtract bias from exponent
  - – Single precision uses bias of 127; DP uses 1023
- 0 uses $0000\ 0000_{\text{two}}$ => 0-127 = -127;
  ∞, NaN uses $1111\ 1111_{\text{two}}$ => 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
	. . .	. . .	. . .	. . .
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

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 2.0×10^{-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 $+\infty$, $-\infty$
  - Sign bit 0 or 1, largest exponent, 0 in fraction
- What if do something stupid? ($\infty - \infty$, $0 \div 0$)
  - 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 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

---

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: c.lt.s (eq, ne, lt, le, gt, ge)
    Fl. Pt. Comparison double precision: c.lt.d
  - Fl. Pt. branch: bclt, bclf
- 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

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!
- Floating point is an approximation of reals