CS 61C:
Great Ideas in Computer Architecture
Introduction to Machine Language

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

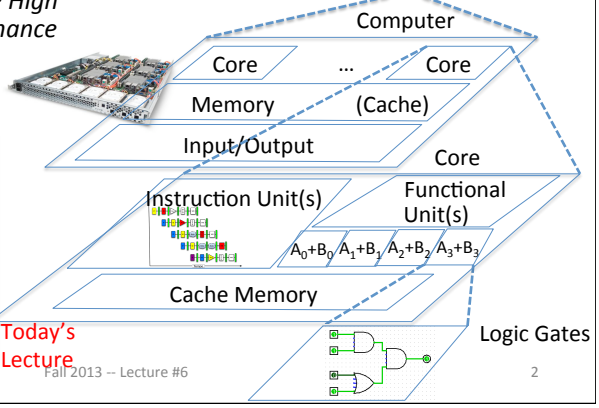
New-School Machine Structures (It's a bit more complicated!)

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

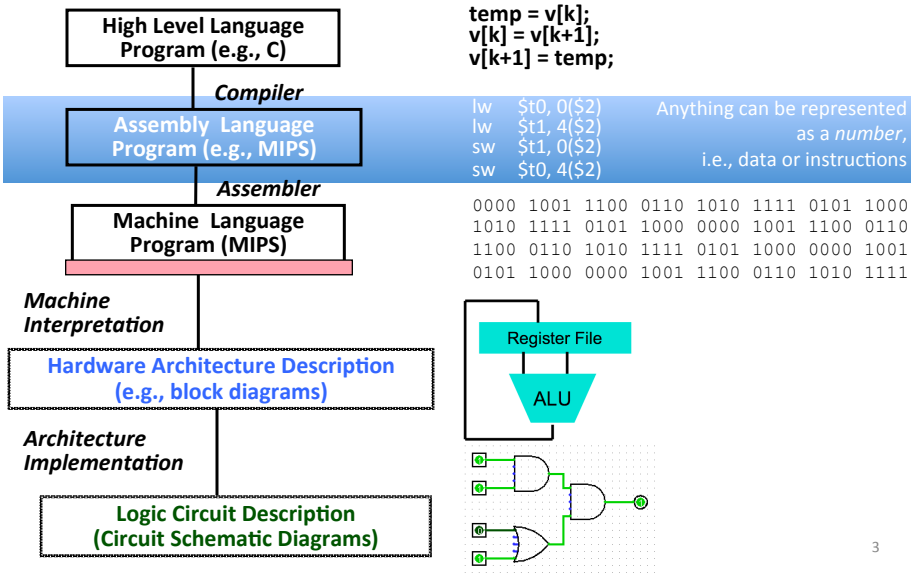
Hardware

Harness Parallelism & Achieve High Performance



Today's Lecture

Levels of Representation/ Interpretation



3

Pointers and Strings in C

- char *p; # p is a pointer to a character
- char x[] = “Randy Katz”; # x points to a literal string
- p = x; # p points to the same place as x
- p = &x[0]; # same as p = x
- *Cannot write* x = “Randy Katz”;
- *Strings are not a primitive type in C (but character arrays are)*
- *Element/character at a time processing possible, but whole string processing requires special routines*

Agenda

- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- And in Conclusion …

Agenda

- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- And in Conclusion …

The Language a Computer Understands

- Word a computer understands: *instruction*
- Vocabulary of all words a computer understands: *instruction set* (aka *instruction set architecture* or *ISA*)
- Different computers may have *different* vocabularies (i.e., different ISAs)
  - iPhone not same as Macbook
- Or the *same* vocabulary (i.e., same ISA)
  - iPhone and iPad computers have same instruction set

The Language a Computer Understands

- Why not all the same? Why not all different? What might be pros and cons?

The Language a Computer Understands

- Why not all the same? Why not all different? What might be pros and cons?
  - Single ISA (*to rule them all*):
    - Leverage common compilers, operating systems, etc.
    - BUT fairly easy to retarget these for different ISAs (e.g., Linux, gcc)
  - Multiple ISAs:
    - Specialized instructions for specialized applications
    - Different tradeoffs in resources used (e.g., functionality, memory demands, complexity, power consumption, etc.)
    - Competition and innovation is good, especially in emerging environments (e.g., mobile devices)

MIPS: Instruction Set for CS 61C

- MIPS is a real-world ISA (see **`www.mips.com`**)
  - Standard instruction set for networking equipment
  - Was also used in original Nintendo-64!
- Elegant example of a *Reduced Instruction Set Computer* (RISC) instruction set
- Invented by John Hennessy @ Stanford
  - *Why not Berkeley/Sun RISC invented by Dave Patterson? Ask him!*

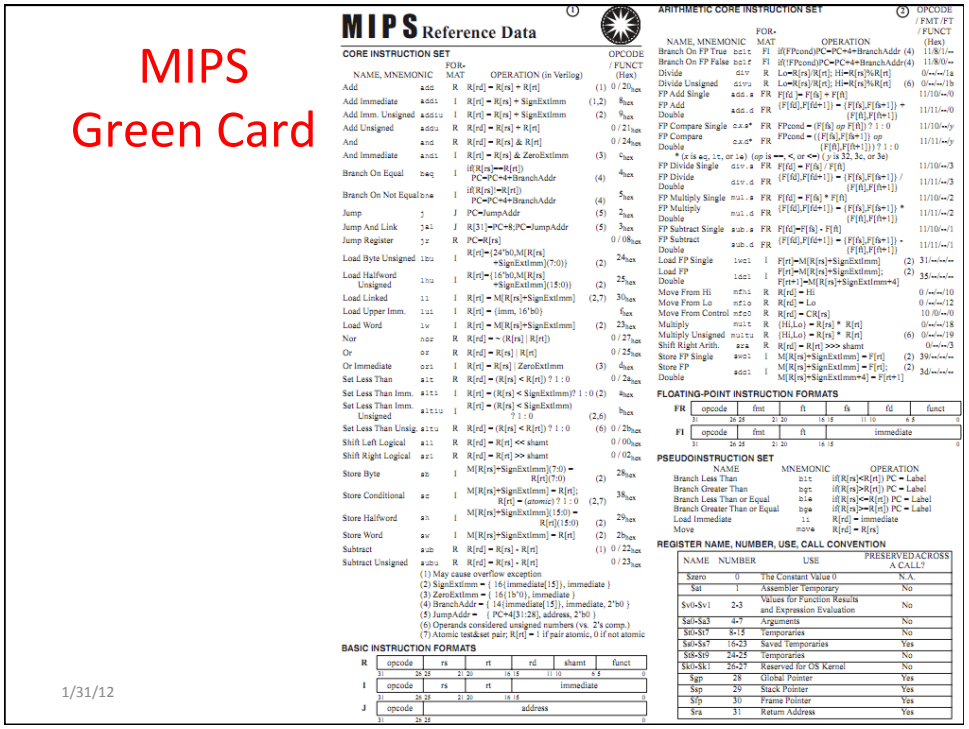
RISC Design Principles

- Basic RISC principle: "A simpler CPU (the hardware that interprets machine language) is a faster CPU" (CPU → *Core*)
- Focus of the RISC design is reduction of the number and complexity of instructions in the ISA
- A number of the more common strategies include:
  - Fixed instruction length, generally a single word;
    Simplifies process of fetching instructions from memory
  - Simplified addressing modes;
    Simplifies process of fetching operands from memory
  - Fewer and simpler instructions in the instruction set;
    Simplifies process of executing instructions
  - Only load and store instructions access memory;
    E.g., no add memory to register, add memory to memory, etc.
  - *Let the compiler do it.* Use a good compiler to break complex high-level language statements into a number of simple assembly language statements

Mainstream ISAs

- ARM (Advanced RISC Machine) is most popular RISC
  - In every smart phone-like device (e.g., iPhone, iPad, iPod, …)
- Intel 80x86 is another popular ISA and is used in Macbook and PCs (Core i3, Core i5, Core i7, …)
  - x86 is a *Complex Instruction Set Computer* (CISC)
  - 20x ARM sold vs. 80x86 (i.e., 5 billion vs. 0.3 billion)

MIPS Green Card



MIPS Reference Data

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 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1'b0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R: opcode (31:26) | rs (25:21) | rt (20:16) | rd (15:11) | shamt (10:6) | funct (5:0)

I: opcode (31:26) | rs (25:21) | rt (20:16) | immediate (15:0)

J: opcode (31:26) | address (25:0)

ARITHMETIC CORE INSTRUCTION SET

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]; (2) F[rt+1]=M[R[rs]+SignExtImm+4]	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]; (2) M[R[rs]+SignExtImm+4] = F[rt+1]	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR: opcode (31:26) | fmt (25:21) | ft (20:16) | fs (15:11) | fd (10:6) | funct (5:0)

FI: opcode (31:26) | fmt (25:21) | ft (20:16) | immediate (15:0)

PSEUDOINSTRUCTION SET

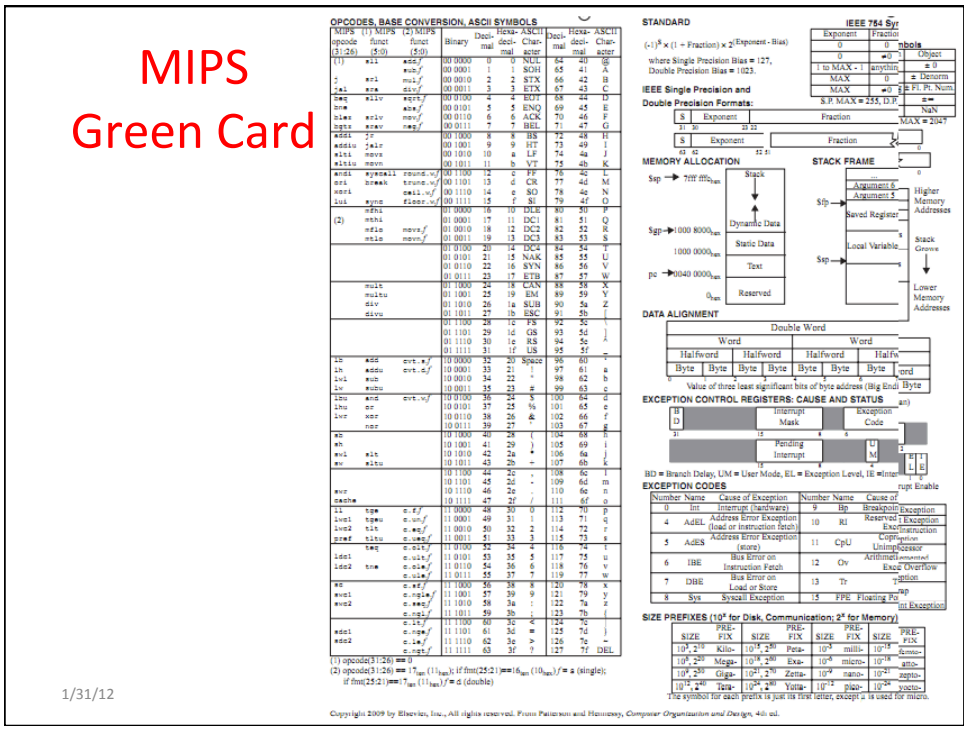
NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

1/31/12

MIPS Green Card



OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.f	00 0000	0	0	NUL	64	40	@
		sub.f	00 0001	1	1	SOH	65	41	A
j	srl	mul.f	00 0010	2	2	STX	66	42	B
jal	sra	div.f	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.f	00 0100	4	4	EOT	68	44	D
bne		abs.f	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.f	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.f	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.f	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.f	00 1101	13	d	CR	77	4d	M
xori		ceil.w.f	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.f	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movs.f	01 0010	18	12	DC2	82	52	R
	mtlo	movn.f	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.f	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.f	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.f	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.f	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.f	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.f	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.f	11 0011	51	33	3	115	73	s
ldc1	teq	c.olt.f	11 0100	52	34	4	116	74	t
		c.ult.f	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.f	11 0110	54	36	6	118	76	v
		c.ule.f	11 0111	55	37	7	119	77	w
sd		c.sf.f	11 1000	56	38	8	120	78	x
swc1		c.ngle.f	11 1001	57	39	9	121	79	y
swc2		c.seq.f	11 1010	58	3a	:	122	7a	z
		c.ngl.f	11 1011	59	3b	;	123	7b	{
		c.lt.f	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.f	11 1101	61	3d	=	125	7d	}
sdc2		c.le.f	11 1110	62	3e	>	126	7e	~
		c.ngt.f	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0
(2) opcode(31:26) == 17_{hex} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) f = s (single); if fmt(25:21)==17_{ten} (11_{hex}) f = d (double)

STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:

	S	Exponent	Fraction	
Single Precision	31	30 to 23	22 to 0	MAX = 254.?
Double Precision	63	62 to 52	51 to 0	

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	± ∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

MEMORY ALLOCATION

STACK FRAME

DATA ALLOCATION

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS

BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interr

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented Exception
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap Exception
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

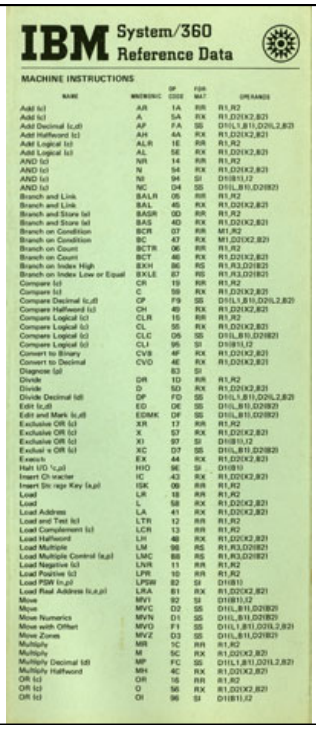
SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
$10^3, 2^{10}$	Kilo-	$10^{15}, 2^{50}$	Peta-	10^{-3}	milli-	10^{-15}	femto-
$10^6, 2^{20}$	Mega-	$10^{18}, 2^{60}$	Exa-	10^{-6}	micro-	10^{-18}	atto-
$10^9, 2^{30}$	Giga-	$10^{21}, 2^{70}$	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
$10^{12}, 2^{40}$	Tera-	$10^{24}, 2^{80}$	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

1/31/12

Inspired by the IBM 360 "Green Card"



MIPS Instructions

- Every computer does arithmetic
- *Instruct* a computer to do addition:

```
add a, b, c
```

  - Add `b` to `c` and put sum into `a`
- 3 operands: 2 sources + 1 destination for sum
- One operation per MIPS instruction
- How do you write the same operation in C?

Guess More MIPS instructions

- Subtract `c` from `b` and put difference in `a`?
- Multiply `b` by `c` and put product in `a`?
- Divide `b` by `c` and put quotient in `a`?

Guess More MIPS instructions

- Subtract `c` from `b` and put difference in `a`?

```
sub a, b, c
```

- Multiply `b` by `c` and put product in `a`?

```
mul a, b, c
```

- Divide `b` by `c` and put quotient in `a`?

```
div a, b, c
```

Guess More MIPS instructions

- C operator &: `c & b` with result in `a`?
- C operator |: `c | b` with result in `a`?
- C operator <<: `b << c` with result in `a`?
- C operator >>: `b >> c` with result in `a`?

Guess More MIPS instructions

- C operator &: `c & b` with result in `a`?
  `and a, b, c`
- C operator |: `c | b` with result in `a`?
  `or a, b, c`
- C operator <<: `b << c` with result in `a`?
  `sll a, b, c`
- C operator >>: `b >> c` with result in `a`?
  `srl a, b, c`

Example Instructions

- MIPS instructions are inflexible, rigid:
  - Just one arithmetic operation per instruction
  - Always with three operands
- How write this C expression in MIPS?

```
a = b + c + d + e
```


Example Instructions

- MIPS instructions are inflexible, rigid:
  - Just one arithmetic operation per instruction
  - Always with three operands
- How write this C expression in MIPS?

```
a = b + c + d + e
```

```
add t1, d, e
add t2, c, t1
add a, b, t2
```

Comments in MIPS

- Can add comments to MIPS instruction by putting # that continues to end of line of text

```
add a, b, c # b + c is placed in a
add a, a, d # b + c + d is now in a
add a, a, e # b + c + d + e is in a
```

- Are *extremely* useful!

C to MIPS

- What is MIPS code that performs same as?

```
a = b + c;
d = a - e;
```

- What is MIPS code that performs same as?

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

C to MIPS

- What is MIPS code that performs same as?

```
a = b + c;  add a, b, c
d = a – e;  sub d, a, e
```

- What is MIPS code that performs same as?

```
f = (g + h) – (i + j);
add t1, i, j
add t2, g, h
sub f, t2, t1
```


For a given function, which programming language likely takes the most lines of code? (most to least)

- ☐ Python, MIPS, C
- ☐ C, Python, MIPS
- ☐ MIPS, Python, C
- ☐ MIPS, C, Python

For a given function, which programming language likely takes the most lines of code? (most to least)

- ☐ Python, MIPS, C
- ☐ C, Python, MIPS
- ☐ MIPS, Python, C
- ☐ MIPS, C, Python

27

Agenda

- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- And in Conclusion …

Administrivia

- This week in lab and homework:
  - HW #2 due Sunday
  - Lab #3 EC2 to be posted soon

CS61c in the News

Agenda

- Machine Language
- Administrivia
- Operands
- Technology Break
- Decisions
- Summary

Computer Hardware Operands

- High-Level Programming languages: could have millions of variables
- Instruction sets have fixed, smaller number
- Called *registers*
  - "Bricks" of computer hardware
  - Fastest way to store data in computer hardware
  - Visible to (the "assembly language") programmer
- MIPS Instruction Set has 32 registers

Why Just 32 Registers?

- RISC Design Principle: *Smaller is faster*
  - But you can be too small …
- Hardware would likely be slower with 64, 128, or 256 registers
- 32 is enough for compiler to translate typical C programs, and not run out of registers very often
  - ARM instruction set has only 16 registers
  - May be faster, but compiler may run out of registers too often (aka "spilling registers to memory")

Names of MIPS Registers

- For registers that hold programmer variables:
  `$s0, $s1, $s2, …`
- For registers that hold temporary variables:
  `$t0, $t1, $t2, …`

Names of MIPS Registers

- Suppose variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is MIPS for

```
f = (g + h) – (i + j);
```


Names of MIPS Registers

- Suppose variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is MIPS for

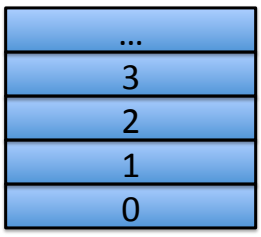
```
f = (g + h) – (i + j);
  add $t1, $s3, $s4
  add $t2, $s1, $s2
  sub $s0, $t2, $t1
```

Size of Registers

- *Bit* is the atom of Computer Hardware: contains either 0 or 1
  - True "alphabet" of computer hardware is 0, 1
  - Will eventually express MIPS instructions as combinations of 0s and 1s (in Machine Language)
- MIPS registers are 32 bits wide
- MIPS calls this quantity a *word*
  - Some computers use 16-bit or 64-bit wide words
  - E.g., Intel 80x86, MIPS64

Data Structures vs. Simple Variables

- In addition to registers, a computer also has *memory* that holds millions / billions of words
- Memory is a single dimension array, starting at 0
- To access memory, need an *address* (like an array index)
- But MIPS instructions only operate on registers!
- Solution: instructions specialized to transfer words (data) between memory and registers
- Called *data transfer instructions*



Transfer from Memory to Register

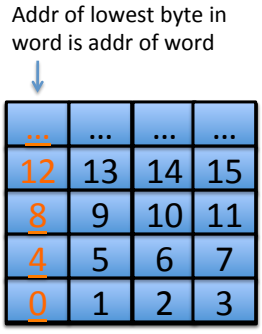
- MIPS instruction: *Load Word*, abbreviated `lw`
- Assume A is an array of 100 words, variables `g` and `h` map to registers `$s1` and `$s2`, the starting address/base address of the array A is in `$s3`
- ```
  int  A[100];
  g = h + A[3];
  ```
- Becomes:

```
lw $t0,3($s3)     # Temp reg $t0 gets A[3]
add $s1,$s2,$t0   # g = h + A[3]
```


Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit item is called a *byte*

  (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as leftmost byte


```

Transfer from *Memory* to *Register*

- MIPS instruction: *Load Word*, abbreviated `lw`
- Assume A is an array of 100 words, variables `g` and `h` map to registers `$s1` and `$s2`, the starting address/base address of the array A is in `$s3`

```
g = h + A[3];
```

- Becomes:

```
lw $t0,12($s3)   # Temp reg $t0 gets A[3]
add $s1,$s2,$t0  # g = h + A[3]
```


Transfer from *Register* to *Memory*

- MIPS instruction: *Store Word*, abbreviated `sw`
- Assume A is an array of 100 words, variables `g` and `h` map to registers `$s1` and `$s2`, the starting address, or base address, of the array A is in `$s3`

```
A[10] = h + A[3];
```

- Turns into

```
lw  $t0,12($s3)  # Temp reg $t0 gets A[3]
add $t0,$s2,$t0  # t0 = h + A[3]
                 # A[10] = h + A[3]
```

Transfer from *Register* to *Memory*

- MIPS instruction: *Store Word*, abbreviated `sw`
- Assume A is an array of 100 words, variables `g` and `h` map to registers `$s1` and `$s2`, the starting address, or base address, of the array A is in `$s3`

```
A[10] = h + A[3];
```

- Turns into

```
lw  $t0,12($s3)  # Temp reg $t0 gets A[3]
add $t0,$s2,$t0  # t0 = h + A[3]
sw  $t0,40($s3)  # A[10] = h + A[3]
```


Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 8 GB on laptop)
- and the RISC principle is…
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!

Which of the following is TRUE?

- `add $t0,$t1,4($t2)` is valid MIPS
- Can byte address 8GB with a MIPS word
- `imm` must be a multiple of 4 for `lw $t0,imm($s0)` to be valid
- If MIPS halved the number of registers available, it would be twice as fast

45

Which of the following is TRUE?
NONE!

- `add $t0,$t1,4($t2)` is valid MIPS
- Can byte address 8GB with a MIPS word
- `imm` must be a multiple of 4 for `lw $t0,imm($s0)` to be valid
- If MIPS halved the number of registers available, it would be twice as fast

46

And In Conclusion …

- Computer words and vocabulary are called instructions and instruction set respectively
- MIPS is example RISC instruction set in this class
- Rigid format: one operation, two source operands, one destination
  - add, sub, mul, div, and, or, sll, srl
  - lw, sw to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, if-then-else in C to MIPS instructions