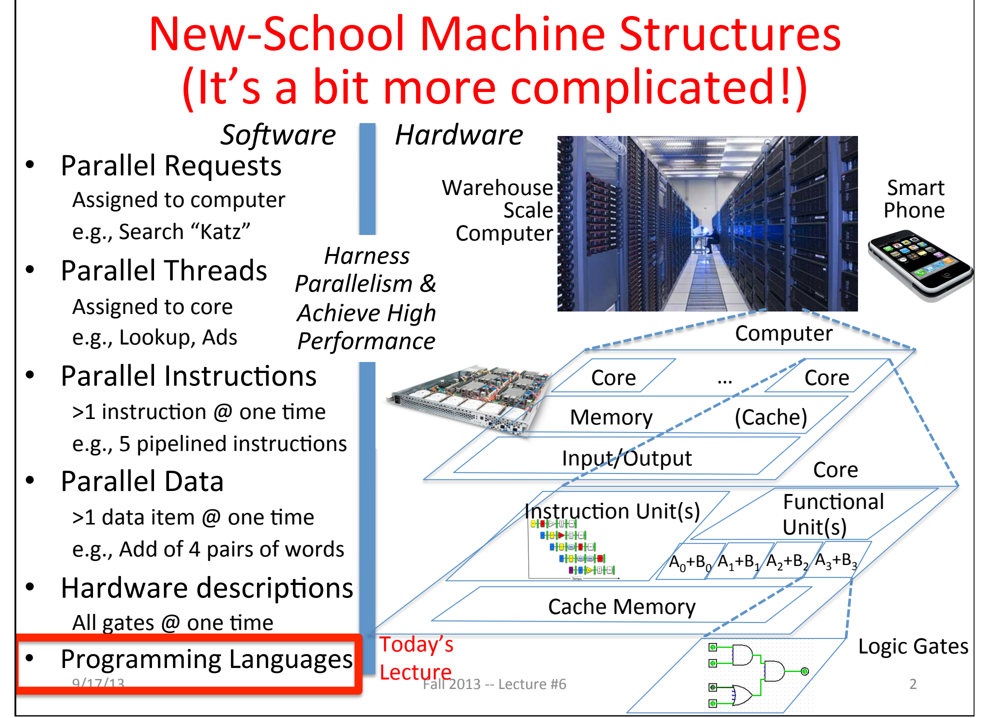
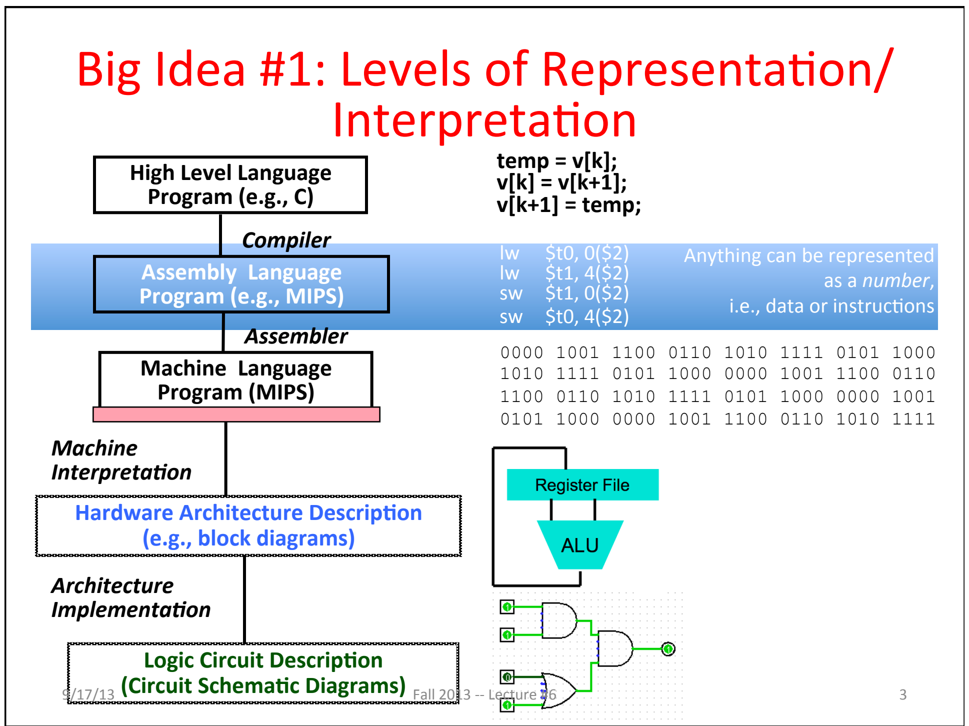
CS 61C:
Great Ideas in Computer Architecture
More MIPS Machine Language

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12





Review

- Computer words and vocabulary are called *instructions* and *instruction set* respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 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

Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- Technology Break
- Functions
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- Technology Break
- Functions
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

Computer Decision Making

- Based on computation, do something different
- In programming languages: if-statement
  - Sometimes combined with gotos and labels
- MIPS: if-statement instruction is

  `beq register1,register2,L1`

  means go to statement labeled L1
  if value in register1 = value in register2
  (otherwise, go to next statement)
- `beq` stands for *branch if equal*
- Other instruction: `bne` for *branch if not equal*

Example If Statement

- Assuming translations below, compile if block

  f → `$s0` g → `$s1` h → `$s2`

  i → `$s3` j → `$s4`

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

- May need to negate branch condition

Example If Statement

- Assuming translations below, compile if block

  f → `$s0` g → `$s1` h → `$s2`

  i → `$s3` j → `$s4`

```
if (i == j)          bne $s3,$s4,Exit
  f = g + h;         add $s0,$s1,$s2
               Exit:
```

- May need to negate branch condition

Types of Branches

- **Branch** – change of control flow
- **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (`beq`) or branch *if not* equal (`bne`)
- **Unconditional Branch** – always branch
  - a MIPS instruction for this*: jump (`j`)*

Making Decisions in C or Java

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

- If false, skip over "then" part to "else" part
  => use conditional branch `bne`
- Otherwise, (its true) do "then" part and skip over "else" part
  => use unconditional branch `j`

Making Decisions in MIPS

- Assuming translations below, compile
  f → `$s0` g → `$s1` h → `$s2`
  i → `$s3` j → `$s4`

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

Which of the following is FALSE?

- ☐ Can make an unconditional branch from a conditional branch instruction
- ☐ Can make a loop with `j`
- ☐ Can make a loop with `beq`
- ☐ Can always return from a function call with `j`

15

Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- Technology Break
- Functions
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

9/17/13 Fall 2013 -- Lecture #6 16

Strings: C vs. Java

- Recall: a string is just a long sequence of characters (i.e., array of chars)
- C: 8-bit ASCII, define strings with end of string character NUL (0 in ASCII)
- Java: 16-bit Unicode, first entry gives length of string

Strings

- "Cal" in ASCII in C; How many bytes?
- Using 1 integer per byte, what does it look like?

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Strings

- "Cal" in Unicode in Java; How many bytes?
- Using 1 integer per byte, what does it look like? (For Latin alphabet, 1st byte is 0, 2nd byte is ASCII)

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Support for Characters and Strings

- Load a word, use `andi` to isolate byte

```
lw   $s0,0($s1)
andi $s0,$s0,255  # Zero everything but last 8 bits
```

- RISC Design Principle: "Make the Common Case Fast"—Many programs use text: MIPS has *load byte* instruction (`lb`)

```
lb $s0,0($s1)
```

- Also *store byte* instruction (`sb`)

Support for Characters and Strings

- Load a word, use `andi` to isolate half of word

```
lw   $s0,0($s1)
andi $s0,$s0,65535 # Zero everything but last 16 bits
```

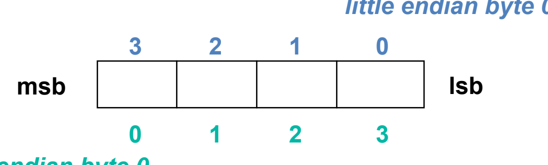
- RISC Design Principle #3: "Make the Common Case Fast"—Many programs use text, MIPS has *load halfword* instruction (`lh`)

```
lh $s0,0($s1)
```

- Also *store halfword* instruction (`sh`)

Endianess

- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



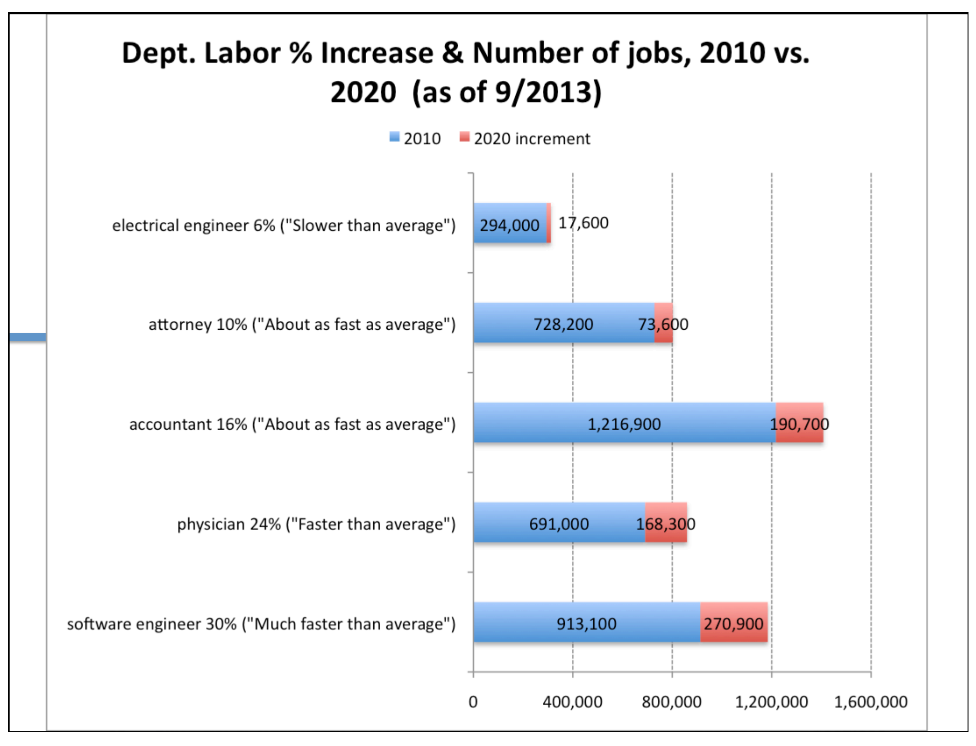
- Big Endian: address of most significant byte = word address (xx00 = "Big End" of word): IBM 360/370, MIPS, Sparc
- Little Endian: address of least significant byte = word address (xx00 = "Little End" of word): Intel 80x86
- Can only tell if access same data with load/store byte and load/store word

Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- Technology Break
- Functions
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

Administrivia

- This week in lab and homework:
  – Lab #3 EC2
  – HW #3 Posted
  – Project #1 posted



Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- Functions
- Technology Break
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

9/17/13 Fall 2013 -- Lecture #6 28

Fast String Copy Code in C

- Copy x[] to y[]

```
char *p, *q;
  p = &x[0];   /* p = x */
     /* set p to address of 1st char of x */
  q = &y[0]; /* q = y also OK */
     /* set q to address of 1st char of y */
  while((*q++ = *p++) != '\0') ;
```


Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
                                    # $t1 = &p (BA), q @ &p + 4
                                    # $s1 = p
                                    # $s2 = q
Loop:                               # $t2 = *p
                                    # *q = $t2
                                    # p = p + 1
                                    # q = q + 1
                                    # if *p == 0, go to Exit
      j Loop                        # go to Loop
Exit:  #  N characters => N*6 + 3 instructions
```

Fast String Copy in MIPS Assembly

Get addresses of x and y into `$s1`, `$s2`

`p` and `q` are assigned to these registers

```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)          # $s1 = p
      lw $s2,4($t1)          # $s2 = q
Loop: lb $t2,0($s1)          # $t2 = *p
      sb $t2,0($s2)          # *q = $t2
      addi $s1,$s1,1         # p = p + 1
      addi $s2,$s2,1         # q = q + 1
      beq $t2,$zero,Exit     # if *p == 0, go to Exit
      j Loop                 # go to Loop
Exit: # N characters => N*6 + 3 instructions
```

BA+4 → Q
BA → P

9/17/13 Fall 2013 -- Lecture #6 38

Student Roulette?

Which statement is TRUE?

```
char *p, *q;
  p = &x[0];  q = &y[0];
  while((*q++ = *p++) != '\0') ;
```

- ☐ `$t1` corresponds to `p`
- ☐ `$s1` corresponds to `p`
- ☐ `$s1` corresponds to `q`
- ☐ `$s1` corresponds to `*p`

```
      lw $t1, Base Address
      lw $s1,0($t1)
      lw $s2,4($t1)
Loop: lb $t2,0($s1)
      sb $t2,0($s2)
      addi $s1,$s1,1
      addi $s2,$s2,1
      beq $t2,$zero,Exit
      j Loop
Exit:
```

40

Assembler Pseudo-instructions

- Register $zero always contains 0
- Can use "pseudo-instruction" in assembly language to make it programming easier
- Example

```
clear $rt
```

- Implemented as:

```
add $rt, $zero, $zero
```


More Pseudo-Instructions

Name	instruction syntax	Real instruction translation
Move	move $rt,$rs	addi $rt,$rs,0
Clear	clear $rt	add $rt,$zero,$zero
Load Address	la $rd, LabelAddr	lui $rd, LabelAddr[31:16]; ori $rd,$rd, LabelAd
Load Immediate	li $rd, IMMED[31:0]	lui $rd, IMMED[31:16]; ori $rd,$rd, IMMED[15
Branch unconditionally	b Label	beq $zero,$zero,Label
Branch and link	bal $rs,Label	bgezal $zero,Label
Branch if greater than	bgt $rs,$rt,Label	slt $at,$rt,$rs; bne $at,$zero,Label
Branch if less than	blt $rs,$rt,Label	slt $at,$rs,$rt; bne $at,$zero,Label
Branch if greater than or equal	bge $rs,$rt,Label	slt $at,$rs,$rt; beq $at,$zero,Label
Branch if less than or equal	ble $rs,$rt,Label	slt $at,$rt,$rs; beq $at,$zero,Label
Branch if greater than unsigned	bgtu $rs,$rt,Label	
Branch if greater than zero	bgtz $rs,$rt,Label	
Multiplies and returns only first 32 bits	mul $d, $s, $t	mult $s, $t; mflo $d

Assembler Pseudo-instructions

- See http://en.wikipedia.org/wiki/MIPS_architecture
- Load Address (asm temp regs): $at = Label Address

```
la $at, LabelAddr
```

- Implemented as:

```
lui $at, LabelAddr[31:16];
ori $at, $at, LabelAddr[15:0]
```


Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- **Technology Break**
- Functions
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

Agenda

- Decisions
- Strings
- Administrivia
- String Copy Example
- Technology Break
- Functions
- MIPS register allocation convention
- *Memory Heap*
- And in Conclusion, …

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling program can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

MIPS Function Call Conventions

- Registers way faster than memory, so use registers
- `$a0–$a3`: four *argument* registers to pass parameters
- `$v0–$v1`: two *value* registers to return values
- `$ra`: one *return address* register to return to the point of origin
- (7 + $zero +$at of 32, 23 left!)

MIPS Registers Assembly Language Conventions

- $t0-$t9: 10 x temporaries (intermediates)
- $s0-$s7: 8 x "saved" temporaries (program variables)
- 18 registers
- 32 – (18 + 9) = 5 left

MIPS Function Call Instructions

- Invoke function: *jump and link* instruction (`jal`)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of following instruction in register `$ra`

```
jal ProcedureAddress
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register

```
jr $ra
```


Notes on Functions

- Calling program (*caller*) puts parameters into registers `$a0–$a3` and uses `jal X` to invoke X (*callee*)
- Must have register in computer with address of currently executing instruction
  - Instead of Instruction Address Register (better name), historically called *Program Counter* (*PC*)
  - It's a program's counter, it doesn't count programs!
- `jr $ra` puts address inside `$ra` into PC
- What value does `jal X` place into `$ra`? **Next PC**
  PC + 4

Where Save Old Registers Values to Restore Them After Function Call

- Need a place to place old values before call function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `$sp` is the *stack pointer* in MIPS
- Convention is grow from high to low addresses
  - Push decrements `$sp`, Pop increments `$sp`
- (28 out of 32, 4 left!)

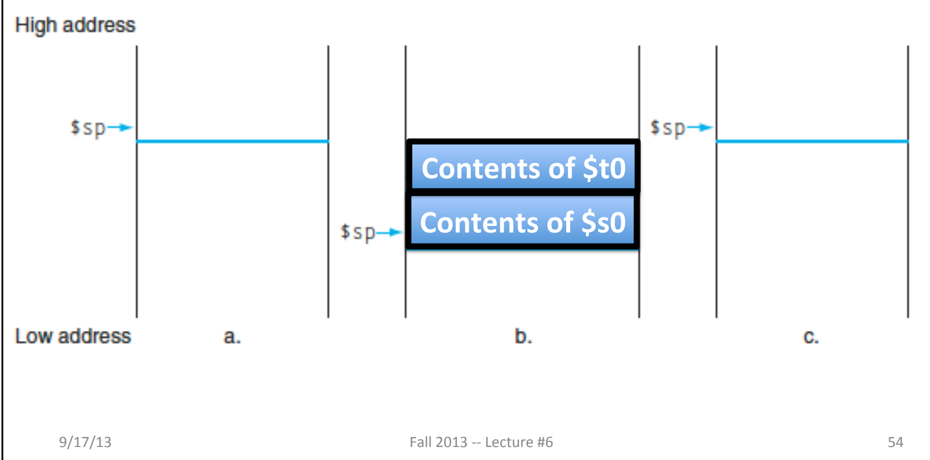
Example

```
int leaf_example
  (int g, int h, int i, int j)
{
  int f;
  f = (g + h) – (i + j);
  return f;
}
```

- Parameter variables `g`, `h`, `i`, and `j` in argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` in `$s0`
- Assume need one temporary register `$t0`

Stack Before, During, After Function

- Need to save old values of `$s0` and `$t0`



MIPS Code for leaf_example

```
leaf_example:
    # adjust stack for 2 int items
    # save $t0 for use afterwards
    # save $s0 for use afterwards
    # f = g + h
    # t0 = i + j
    # return value (g + h) – (i + j)
    # restore $s0 for caller
    # restore $t0 for caller
    # delete 2 items from stack
    # jump back to calling routine
```

MIPS Code for leaf_example

leaf_example:

```
addi $sp,$sp,-8   # adjust stack for 2 int items
sw $t0, 4($sp)    # save $t0 for use afterwards
sw $s0, 0($sp)    # save $s0 for use afterwards
add $s0,$a0,$a1   # f = g + h
add $t0,$a2,$a3   # $t0 = i + j
sub $v0,$s0,$t0   # return value (g + h) – (i + j)
lw $s0, 0($sp)    # restore register $s0 for caller
lw $t0, 4($sp)    # restore register $t0 for caller
addi $sp,$sp,8    # adjust stack to delete 2 items
jr $ra            # jump back to calling routine
```


What will the printf output?

- ☐ Print -4
- ☐ Print 4
- ☐ a.out will crash
- ☐ None of the above

```
static int *p;
int leaf (int g, int h,
  int i, int j)
{
  int f; p = &f;
  f = (g + h) – (i + j);
  return f;
}
int main(void) { int x;
  x = leaf(1,2,3,4);
  x = leaf(3,4,1,2);
  …
  printf("%d\n",*p);
}
```

And in Conclusion, …

- C is function oriented; code reuse via functions
  - Jump and link (`jal`) invokes, jump register (`jr $ra`) returns
  - Registers `$a0-$a3` for arguments, `$v0-$v1` for return values
- Stack for spilling registers, nested function calls, C local (automatic) variables