CS 61C:
Great Ideas in Computer Architecture
Strings and Functions

Instructor:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/sp12

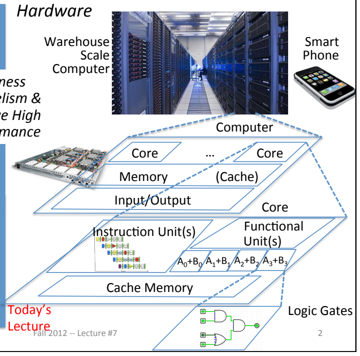
New-School Machine Structures (It's a bit more complicated!)

Software

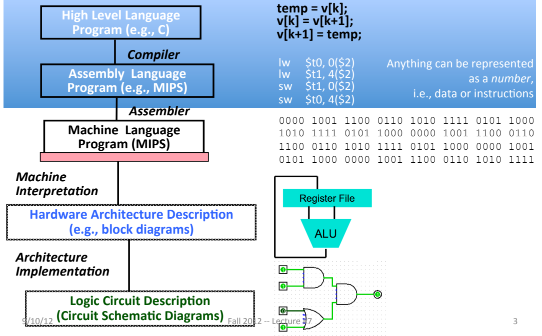
- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions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

Harness Parallelism & Achieve High Performance

Today's Lecture



Big Idea #1: Levels of Representation/ Interpretation



```
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

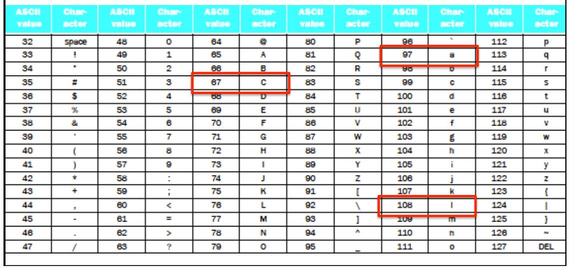
- Review
- Strings in C and MIPS
- Administrivia
- Functions
- And in Conclusion, …

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

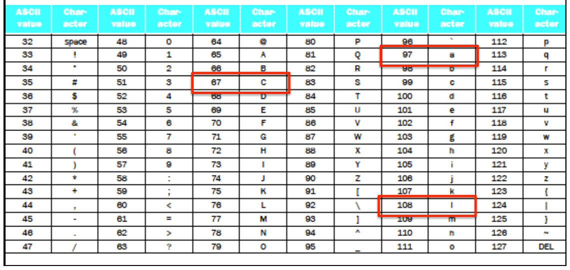
- "Cal" in ASCII in C; How many bytes?
- Using 1 integer per byte, what does it look like?



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

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*

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`)

Fast String Copy Code in C

- Copy x[] to y[]

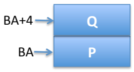
```
char *p, *q;
  p = &x[0];  /* p = x */
     /* set p to address of 1st char of x */
  q = &y[0]; /* q = y also OK */
     /* set q to address of 1st char of y */
  while((*q++ = *p++) != ‘\0’) ;
```

Fast String Copy in MIPS Assembly

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

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

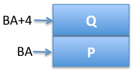
```
                          # $t1 = &p (BA), q @ &p + 4
                          # $s1 = p
                          # $s2 = q
Loop:                     # $t2 = *p
                          # *q = $t2
                          # p = p + 1
                          # q = q + 1
                          # 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



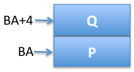
```
      lw $t1, Base Address (e.g., BA)
      lw $s1,0($t1)          # $s1 = p
      lw $s2,4($t1)          # $s2 = q
Loop: lb $t2,0($s1)          # $t2 = *p
      sb $t2,0($s2)          # *q = $t2
      addi $s1,$s1,1         # p = p + 1
      addi $s2,$s2,1         # q = q + 1
                  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

Student Roulette?

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
```

Student Roulette?

Which statement is TRUE?

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

```
      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:
```

☐ `$t1` corresponds to p

☐ `$s1` corresponds to p

☐ `$s1` corresponds to q

☐ `$s1` corresponds to *p

Agenda

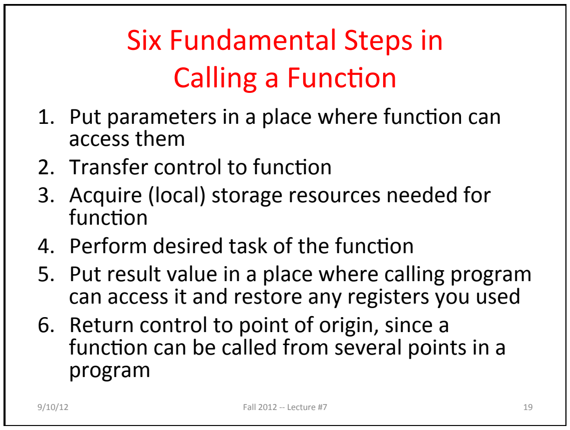
- Review
- Strings in C and MIPS
- Administrivia
- Functions
- And in Conclusion, …

Administrivia

- Map-Reduce Project #1
  - Two Parts, first part due Sunday
  - Write-up posted last night (thanks Alan!)
- Lab #3
  - Hands on EC2, needed for Project #1, *Part 2*
- HW #3
  - C practice/numbers and strings
- … Midterm is coming in < one month!
  - Let us know special accommodation now …
  - CS 188 students, ask your instructors to return my emails!

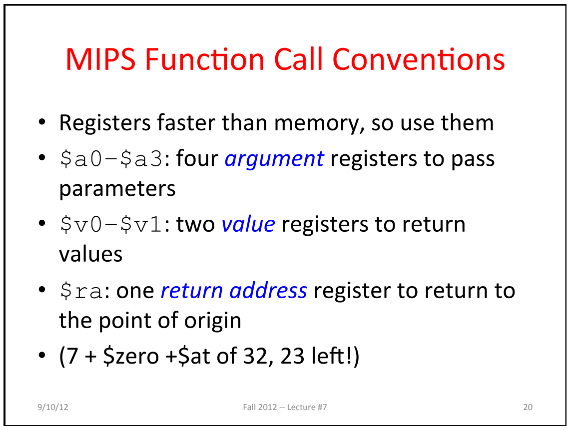
Agenda

- Review
- Strings in C and MIPS
- Administrivia
- Functions
- And in Conclusion, …



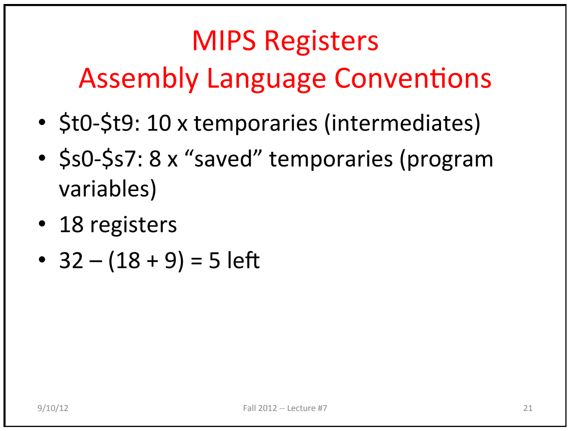
Six Fundamental Steps in Calling a Function

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



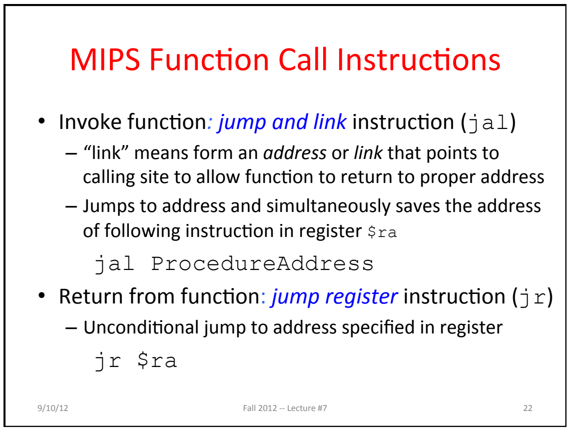
MIPS Function Call Conventions

- Registers faster than memory, so use them
- `$a0–$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



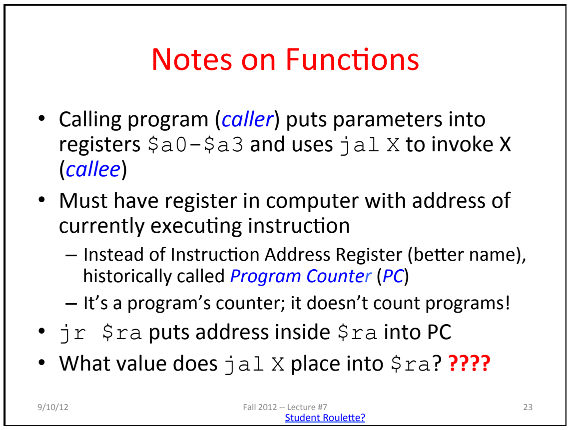
MIPS Function Call Instructions

- 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

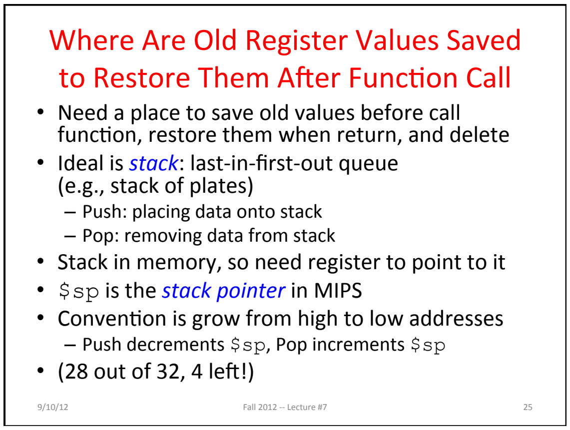
```
jr $ra
```



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`? **????**

Student Roulette?



Where Are Old Register Values Saved to Restore Them After Function Call

- Need a place to save 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!)

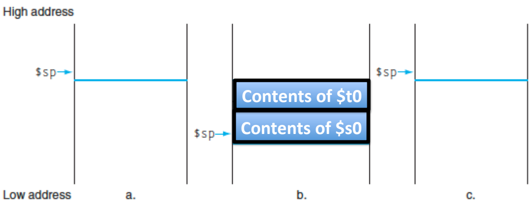
Example

```
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
```

Student Roulette?

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);
}
```

What If a Function Calls a Function? Recursive Function Calls?

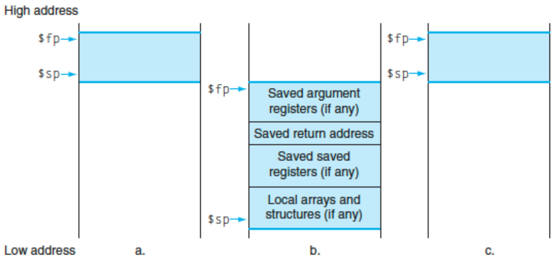
- Would clobber values in `$a0` to `$a3` and `$ra`
- What is the solution?

Student Roulette?

Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits
  - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer (`$fp`) to point to first word of frame
- (29 of 32, 3 left!)

Stack Before, During, After Call



Recursive Function Factorial

```
int fact (int n)
{
  if (n < 1) return (1);
      else return (n * fact(n-1));
}
```

Recursive Function Factorial

```
Fact:
   # adjust stack for 2 items
   addi $sp,$sp,-8
   # save return address
   sw $ra, 4($sp)
   # save argument n
   sw $a0, 0($sp)
   # test for n < 1
   slti $t0,$a0,1
   # if n >= 1, go to L1
   beq $t0,$zero,L1
   # Then part (n==1) return 1
   addi $v0,$zero,1
   # pop 2 items off stack
   addi $sp,$sp,8
   # return to caller
   jr $ra
```

```
L1:
   # Else part (n >= 1)
   # arg. gets (n – 1)
   addi $a0,$a0,-1
   # call fact with (n – 1)
   jal fact
   # return from jal: restore n
   lw $a0, 0($sp)
   # restore return address
   lw $ra, 4($sp)
   # adjust sp to pop 2 items
   addi $sp, $sp,8
   # return n * fact (n – 1)
   mul $v0,$a0,$v0
   # return to the caller
   jr $ra
```

mul is a pseudo instruction

Optimized Function Convention

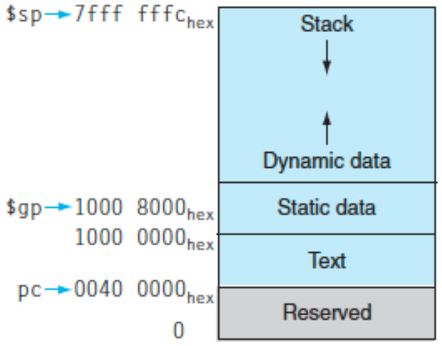
To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

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



Register Allocation and Numbering

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

Which statement is FALSE?

- ☐ MIPS uses `jal` to invoke a function and `jr` to return from a function
- ☐ `jal` saves PC+1 in `%ra`
- ☐ The callee can use temporary registers (`%t`*i*) without saving and restoring them
- ☐ The caller can rely on save registers (`%s`*i*) without fear of callee changing them

And in Conclusion, …

- C strings are char arrays, byte per character, null terminated
- Distinguish pointers and the memory they point to
  - * for dereference, & for address
- 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