Upcoming Calendar

- Lab Due Dates
  - ❖10/16, 10/30, 11/13, 12/4
- Homework Due Dates
  - ❖10/23, 11/6, 11/20, 12/11
- Midterm
  - ❖10/30
- Class Canceled
  - ❖11/25

1

Review

- What is the problem concerning saving registers across function calls in assembly language?
- Who saves $a0?
- Who saves $t0?
- Who saves $s0?
- Who saves $ra?

2

Where are the registers saved?

- Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.
- It would be nice if each function call had its own private memory area.
  - ❖ This would prevent other function calls from overwriting our saved registers—otherwise using memory is no better than using registers.
  - ❖ We could use this private memory for other purposes too, like storing local variables.

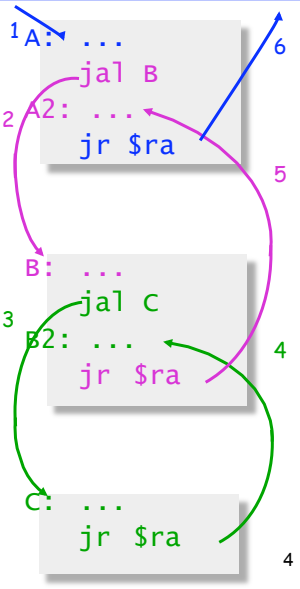
3

Function calls and stacks

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns

- Here, for example, C must return to B *before* B can return to A.



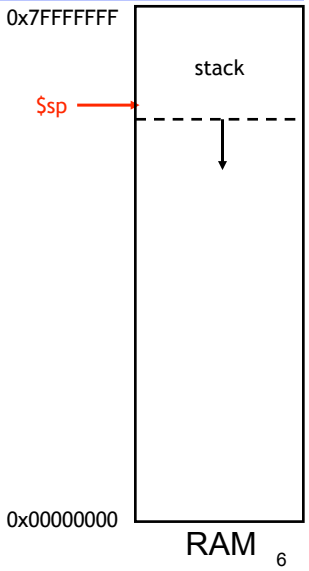
Stacks and function calls

- It's natural to use a stack for function call storage. A block of stack space, called a stack frame, can be allocated for each function call.
  - ❖ When a function is called, it creates a new frame onto the stack, which will be used for local storage.
  - ❖ Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- The stack frame can be used for several purposes.
  - ❖ Caller- and callee-save registers can be put in the stack.
  - ❖ The stack frame can also hold local variables, or extra arguments and return values.

5

The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
  - ❖ The stack grows downward in terms of memory addresses.
  - ❖ The address of the top element of the stack is stored (by convention) in the "stack pointer" register, $sp.
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.



6

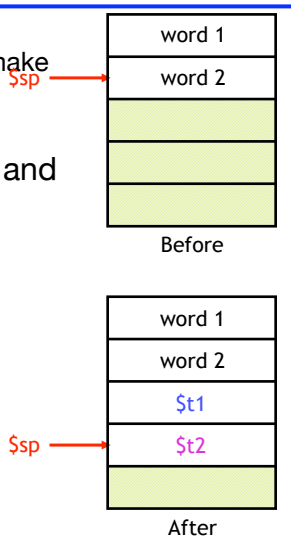
Pushing elements

- To push elements onto the stack:
  - ❖ Move the stack pointer $sp down to make room for the new data.
  - ❖ Store the elements into the stack.
- For example, to push registers $t1 and $t2 onto the stack:

```
addi $sp, $sp, -8
sw   $t1, 4($sp)
sw   $t2, 0($sp)
```

- An equivalent sequence is:

```
sw  $t1, -4($sp)
sw  $t2, -8($sp)
addi $sp, $sp, -8
```



7

Accessing and popping elements

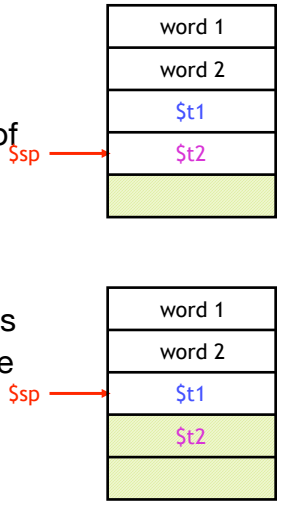
- Any element in the stack can be referenced if you know where it is relative to $sp.
- For example, to retrieve the value of $t1:

```
lw   $s0, 4($sp)
```

- Pop, or "erase," elements by adjusting the stack pointer upwards
- To pop the value of $t2, yielding the stack shown at the bottom:

```
addi $sp, $sp, 4
```

- Popped data is still present in memory, but data past the stack pointer is considered invalid.



8

Representing Strings

- C-style string is represented by an array of bytes
  - ❖ Elements are 1-byte ASCII codes for each character.
  - ❖ A 0 value marks the end of the array.

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

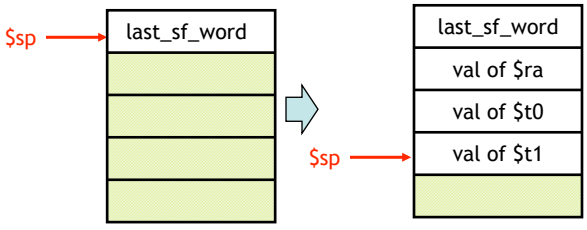
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	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

strlen Example

```
void somefunc() {
  char *str; int a;
  …
  /*uses t0, t1 somewhere */
  …
  a = strlen(str);
  …
}
```

```
int strlen(char *s) {
  int count = 0;
  while (*s != 0) {
    count++;
    s++;
  }
  return count;
}
```

```
somefunc:
  …
  addi $sp, $sp, -12
  sw   $ra, 8($sp)
  sw   $t0, 4($sp)
  sw   $t1, 0($sp)
  add  $a0, $t0, $0
  jal  strlen
  lw   $t1, 0($sp)
  lw   $t0, 4($sp)
  lw   $ra, 8($sp)
  addi $sp, $sp, 12
  …
  jr $ra
```



caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

strlen Example

```
void somefunc() {
  char *str; int a;
  …
  /*uses t0, t1 somewhere */
  …
  a = strlen(str);
  …
}
somefunc:
  …
  addi $sp, $sp, -12
  sw   $ra, 8($sp)
  sw   $t0, 4($sp)
  sw   $t1, 0($sp)
  add  $a0, $t0, $0
  jal  strlen
  lw   $t1, 0($sp)
  lw   $t0, 4($sp)
  lw   $ra, 8($sp)
  addi $sp, $sp, 12
  …
  jr $ra
```

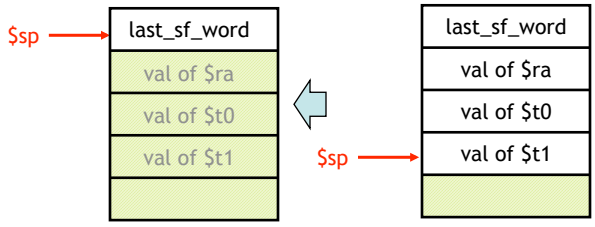
```
int strlen(char *s) {
  int count = 0;
  while (*s != 0) {
    count++;
    s++;
  }
  return count;
}

strlen:
    addi $t0, $0, 0  #count

loop:
    lb   $t1, 0($a0) #get byte
    beq  $t1, $0, end_loop
    addi $t0, $t0, 1 #count++
    addi $a0, $a0, 1 #s++
    j loop

end_loop:
    add $v0, $t0, $0
    jr $ra
```

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

strlen Example

```
void somefunc() {
  char *str; int a;
  …
  /*uses t0, t1 somewhere */
  …
  a = strlen(str);
  …
}
somefunc:
  …
  addi $sp, $sp, -12
  sw   $ra, 8($sp)
  sw   $t0, 4($sp)
  sw   $t1, 0($sp)
  add  $a0, $t0, $0
  jal  strlen
  lw   $t1, 0($sp)
  lw   $t0, 4($sp)
  lw   $ra, 8($sp)
  addi $sp, $sp, 12
  …
  jr $ra
```

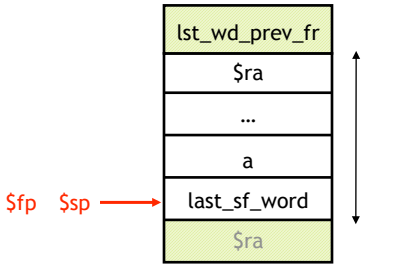
```
int strlen(char *s) {
  int count = 0;
  while (*s != 0) {
    count++;
    s++;
  }
  return count;
}
```



caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

Frame Pointer and Reference

```
void somefunc() {
  char *str; int a;
  …
  /*uses t0, t1 somewhere */
  …
  a = strlen(str);
  …
}
somefunc:
  …
  addi $sp, $sp, -12
  sw   $ra, 8($sp)
  sw   $t0, 4($sp)
  sw   $t1, 0($sp)
  add  $a0, $t0, $0
  jal  strlen
  lw   $t1, 0($sp)
  lw   $t0, 4($sp)
  lw   $ra, 8($sp)
  addi $sp, $sp, 12
  …
  jr $ra
```




```
sw $v0, a_offset($fp)
```

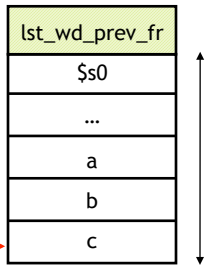
frame pointer

displacement of a from fp == 4

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

Heavyweight Fcns – Set Up Frame

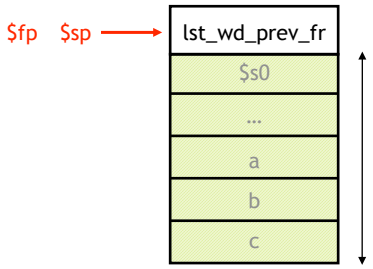
```
void somefunc() {
  int a, b, c;
  …
  a = b + c;
  …
}
somefunc:
    addi $sp, $sp, -48
    sw   $s0, 44($sp)
    sw   $s1, 40($sp)
    ...
    sw   $s7, 16($sp)
    sw   $ra, 12($sp)
    sw   $0,   8($sp)  Initialize a
    sw   $0,   4($sp)  Initialize b
    sw   $0,   0($sp)  Initialize c
    move $fp,  $sp
    ...
```



caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

Heavyweight Fcns – Tear Down Frame

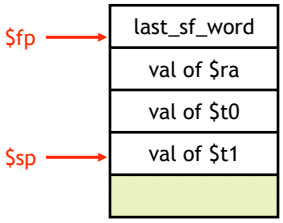
```
void somefunc() {
  int a, b, c;
  …
  a = b + c;
  …
}
somefunc:
 ...
    sw   $s0, 44($sp)
    sw   $s1, 40($sp)
    ...
    sw   $s7, 16($sp)
    sw   $ra, 12($sp)
    addi $sp, $sp, 48
    move $fp, $sp
    jr   $ra
# End of function
```



caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

Caller Saves Registers on Stack

```
void somefunc() {
  …
  /*uses t0, t1 somewhere */
  …
  t2 = small_func(t0);
  …
}
somefunc:
  …
  addi $sp, $sp, -12
  sw   $ra, 8($sp)
  sw   $t0, 4($sp)
  sw   $t1, 0($sp)
  add  $a0, $t0, $0
  jal  small_func
  lw   $t1, 0($sp)
  lw   $t0, 4($sp)
  lw   $ra, 8($sp)
  addi $sp, $sp, 12
  …
  jr $ra
```



caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

Recursive Factorial

```
1 factorial:
2    bgtz $a0, doit          # Argument > 0
3    li   $v0, 1             # Base case, 0! = 1
4    jr   $ra                # Return
5 doit:
6    addi  $sp, sp, -8       # Allocate stack frame
7    sw    $s0,($sp)         # Position for argument n
8    sw    $ra,4($sp)        # Remember return address
9    move $s0, $a0           # Push argument
10   addi  $a0, a0, -1       # Pass n-1
11   jal   factorial         # Figure v0 = (n-1)!
12   mul   $v0,$s0,$v0       # Now multiply by n, v0 = n*(n-1)!
13   lw    $s0,($sp)         # Restore registers from stack
14   lw    $ra,4($sp)        # Get return address
15   addi  $sp, sp, 8        # Pop
16   jr    $ra               # Return
```

17