Calling Conventions

Hakim Weatherspoon
CS 3410, Spring 2012
Computer Science
Cornell University

See P&H 2.8 and 2.12

Goals for Today

Review: Calling Conventions

- call a routine (i.e. transfer control to procedure)
- pass arguments
  - fixed length, variable length, recursively
- return to the caller
  - Putting results in a place where caller can find them
- Manage register

Today

- More on Calling Conventions
- globals vs local accessible data
- callee vs callrer saved registers
- Calling Convention examples and debugging

Goals for Today

Review: Calling Conventions

- call a routine (i.e. transfer control to procedure)
- pass arguments
  - fixed length, variable length, recursively
- return to the caller
  - Putting results in a place where caller can find them
- Manage register

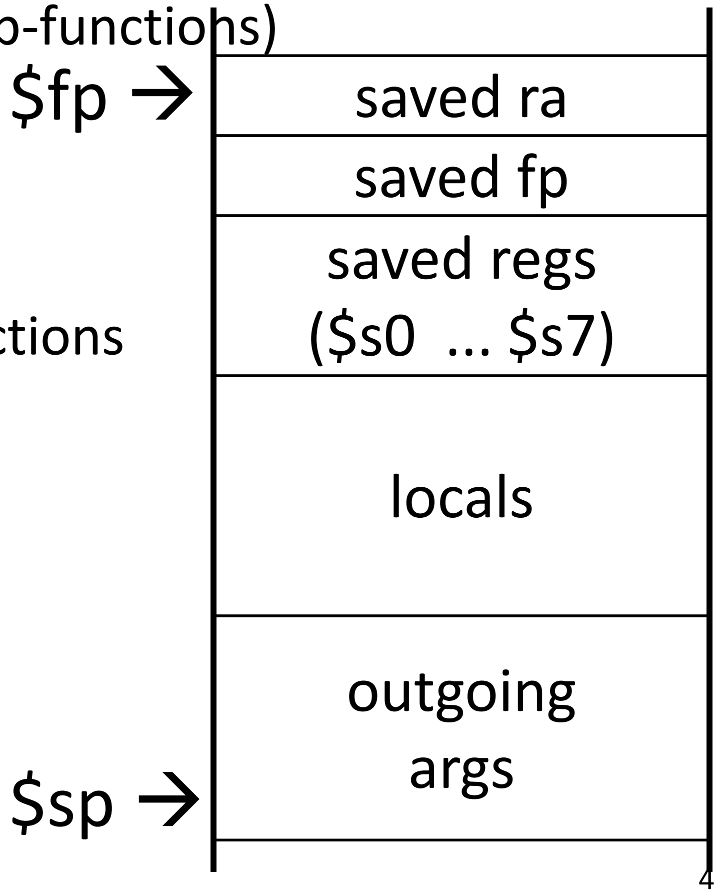
Today

- More on Calling Conventions
- globals vs local accessible data
- callee vs callrer saved registers
- Calling Convention examples and debugging

Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web

Recap: Conventions so far

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent's stack frame
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains $fp
  - contains local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions (i.e. argument "spilling)
  - contains space for first 4 arguments to sub-functions
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp



$fp →

saved ra

saved fp

saved regs ($s0 ... $s7)

locals

outgoing args

$sp →

MIPS Register Conventions

r0	$zero	zero
r1	$at	assembler temp
r2	$v0	function return values
r3	$v1	
r4	$a0	function arguments
r5	$a1	
r6	$a2	
r7	$a3	
r8	$t0	**temps (caller save)**
r9	$t1	
r10	$t2	
r11	$t3	
r12	$t4	
r13	$t5	
r14	$t6	
r15	$t7	

r16	$s0	**saved (callee save)**
r17	$s1	
r18	$s2	
r19	$s3	
r20	$s4	
r21	$s5	
r22	$s6	
r23	$s7	
r24	$t8	**more temps (caller save)**
r25	$t9	
r26	$k0	reserved for kernel
r27	$k1	
r28	$gp	global data pointer
r29	$sp	stack pointer
r30	$fp	frame pointer
r31	$ra	return address

Globals and Locals

Global variables in data segment

- Exist for all time, accessible to all routines

Dynamic variables in heap segment

- Exist between malloc() and free()

Local variables in stack frame

- Exist solely for the duration of the stack frame

Dangling pointers into freed heap mem are bad

Dangling pointers into old stack frames are bad

- C lets you create these, Java does not
- int *foo() { int a; return &a; }

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

Caller-save registers are responsibility of the caller

- Caller-save register values saved only if used after call/return
- The callee function can use caller-saved registers

Callee-save register are the responsibility of the callee

- Values must be saved by callee before they can be used
- Caller can assume that these registers will be restored

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

MIPS ($t0-$t0), x86 (eax, ecx, and edx) are caller-save…

- … a function can freely modify these registers
- … but must assume that their contents have been destroyed if it in turns calls a function.

MIPS $s0 - $s7), x86 (ebx, esi, edi, ebp, esp) are callee-save

- A function may call another function and know that the callee-save registers have not been modified
- However, if it modifies these registers itself, it must restore them to their original values before returning.

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

A caller-save register must be saved and restored around any call to a subprogram.

In contrast, for a callee-save register, a caller need do no extra work at a call site (the callee saves and restores the register if it is used).

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

CALLER SAVED: MIPS calls these temporary registers, $t0-t9

- the calling program saves the registers that it does not want a called procedure to overwrite
- register values are NOT preserved across procedure calls

CALLEE SAVED: MIPS calls these saved registers, $s0-s8

- register values are preserved across procedure calls
- the called procedure saves register values in its AR, uses the registers for local variables, restores register values before it returns.

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

Registers $t0-$t9 are caller-saved registers

- … that are used to hold temporary quantities
- … that need not be preserved across calls

Registers $s0-s8 are callee-saved registers

- … that hold long-lived values
- … that should be preserved across calls

Calling Convention Example

```
int test(int a, int b) {
   int tmp = (a&b)+(a|b);
   int s = sum(tmp,1,2,3,4,5);
   int u = sum(s,tmp,b,a,b,a);
   return u + a + b;
}
```

Calling Convention Example: Prolog, Epilog

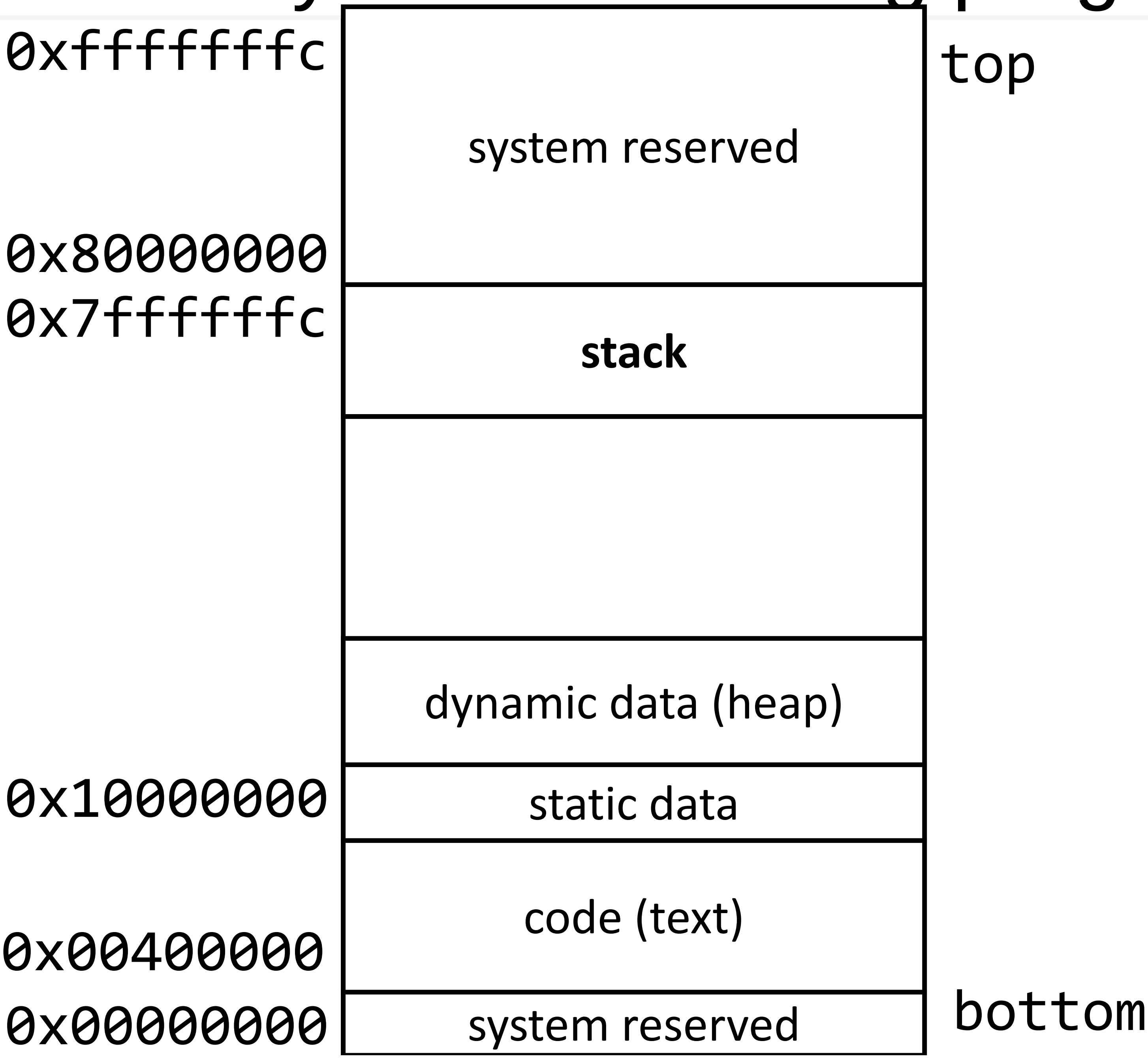
Minimum stack size for a standard function?

Leaf Functions

Leaf function does not invoke any other functions

int f(int x, int y) { return (x+y); }

Anatomy of an executing program



0xfffffffc

top

system reserved

0x80000000

0x7ffffffc

stack

dynamic data (heap)

0x10000000

static data

code (text)

0x00400000

bottom

0x00000000

system reserved

Debugging

init():	0x400000
printf(s, …):	0x4002B4
vnorm(a,b):	0x40107C
main(a,b):	0x4010A0
pi:	0x10000000
str1:	0x10000004

CPU:
$pc=0x004003C0
$sp=0x7FFFFFAC
$ra=0x00401090

	0x00000000
	0x0040010c
	0x0040010a
	0x00000000
	0x00000000
	0x00000000
	0x00000000
	0x004010c4
	0x00000000
	0x00000000
	0x00000015
0x7FFFFFB0	0x10000004
	0x00401090

What func is running?

Who called it?

Has it called anything?

Will it?

Args?

Stack depth?

Call trace?

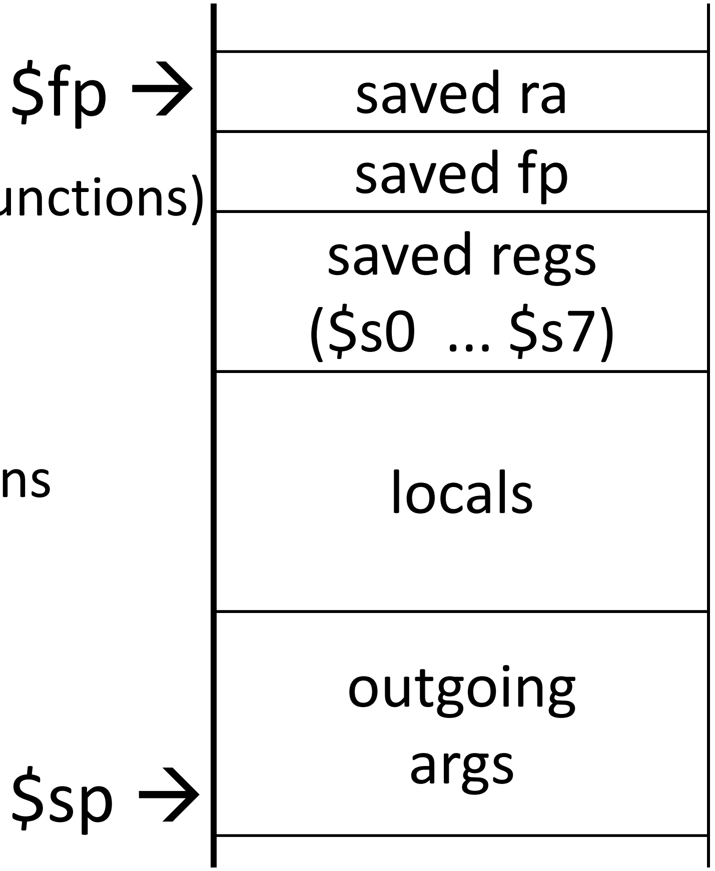
Administrivia

Upcoming agenda

- Schedule PA2 Design Doc Mtg for ***this*** Sunday or Monday
- HW3 due next Tuesday, March 13th
- PA2 Work-in-Progress circuit due before spring break
- Spring break: Saturday, March 17th to Sunday, March 25th
- HW4 due after spring break, before Prelim2
- Prelim2 Thursday, March 29th, right after spring break
- PA2 due Monday, April 2nd, after Prelim2

Recap

- How to write and Debug a MIPS program using calling convention
- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent’s stack frame
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains $fp
  - contains local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions (i.e. argument “spilling)
  - contains space for first 4 arguments to sub-functions
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp



$fp →

saved ra

saved fp

saved regs ($s0 ... $s7)

locals

outgoing args

$sp →