

Calling Conventions

Prof. Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

See P&H 2.8 and 2.12, and A.5-6

Announcement

Upcoming agenda

- PA1 due yesterday
- PA2 available and discussed during lab section this week
- PA2 Work-in-Progress due Monday, March 16th
- PA2 due Thursday, March 26th
- HW2 available next week, due before Prelim2 in April
- **Spring break:** Saturday, March 28th to Sunday, April 5th

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 caller 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 caller saved registers
- Calling Convention examples and debugging

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

MIPS Register Recap

Return address: \$31 (ra)

Stack pointer: \$29 (sp)

Frame pointer: \$30 (fp)

First four arguments: \$4-\$7 (a0-a3)

Return result: \$2-\$3 (v0-v1)

Callee-save free regs: \$16-\$23 (s0-s7)

Caller-save free regs: \$8-\$15, \$24, \$25 (t0-t9)

Reserved: \$26, \$27

Global pointer: \$28 (gp)

Assembler temporary: \$1 (at)

MIPS Register Conventions

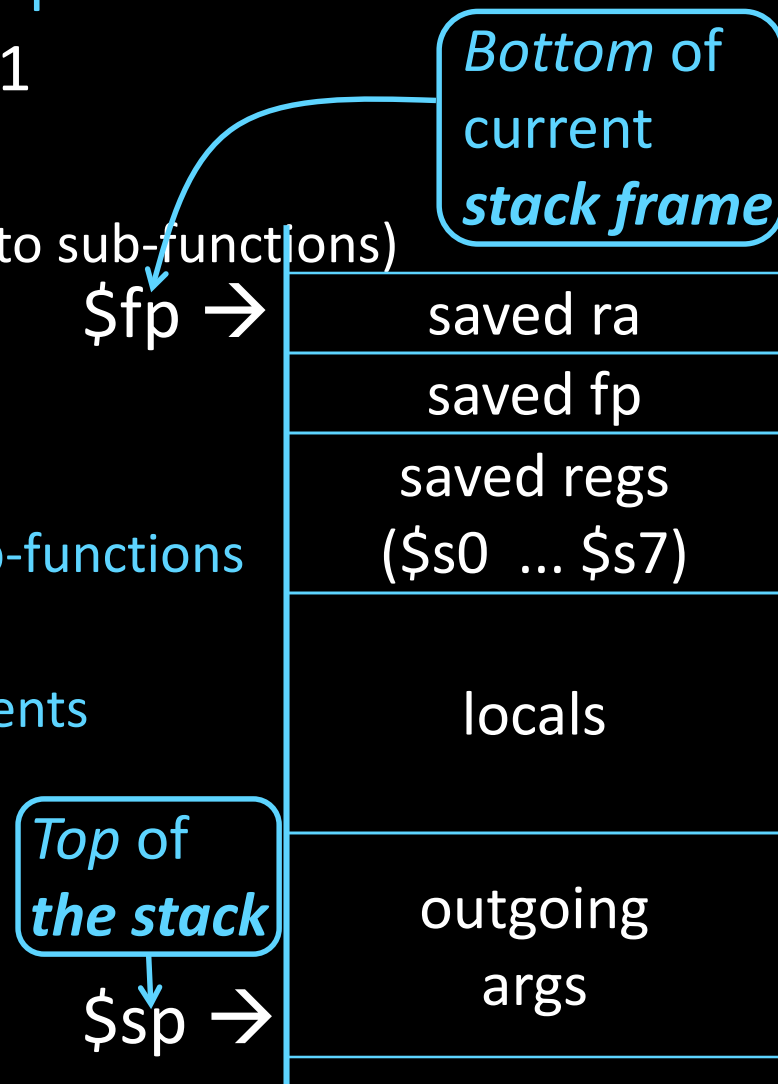
r0	\$zero	zero	r16	\$s0	saved (callee save)
r1	\$at	assembler temp	r17	\$s1	
r2	\$v0	function return values	r18	\$s2	
r3	\$v1		r19	\$s3	
r4	\$a0	function arguments	r20	\$s4	
r5	\$a1		r21	\$s5	
r6	\$a2		r22	\$s6	
r7	\$a3		r23	\$s7	
r8	\$t0	temps (caller save)	r24	\$t8	more temps (caller save)
r9	\$t1		r25	\$t9	reserved for kernel
r10	\$t2		r26	\$k0	
r11	\$t3		r27	\$k1	
r12	\$t4		r28	\$gp	global data pointer
r13	\$t5		r29	\$sp	stack pointer
r14	\$t6		r30	\$fp	frame pointer
r15	\$t7		r31	\$ra	return address

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



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

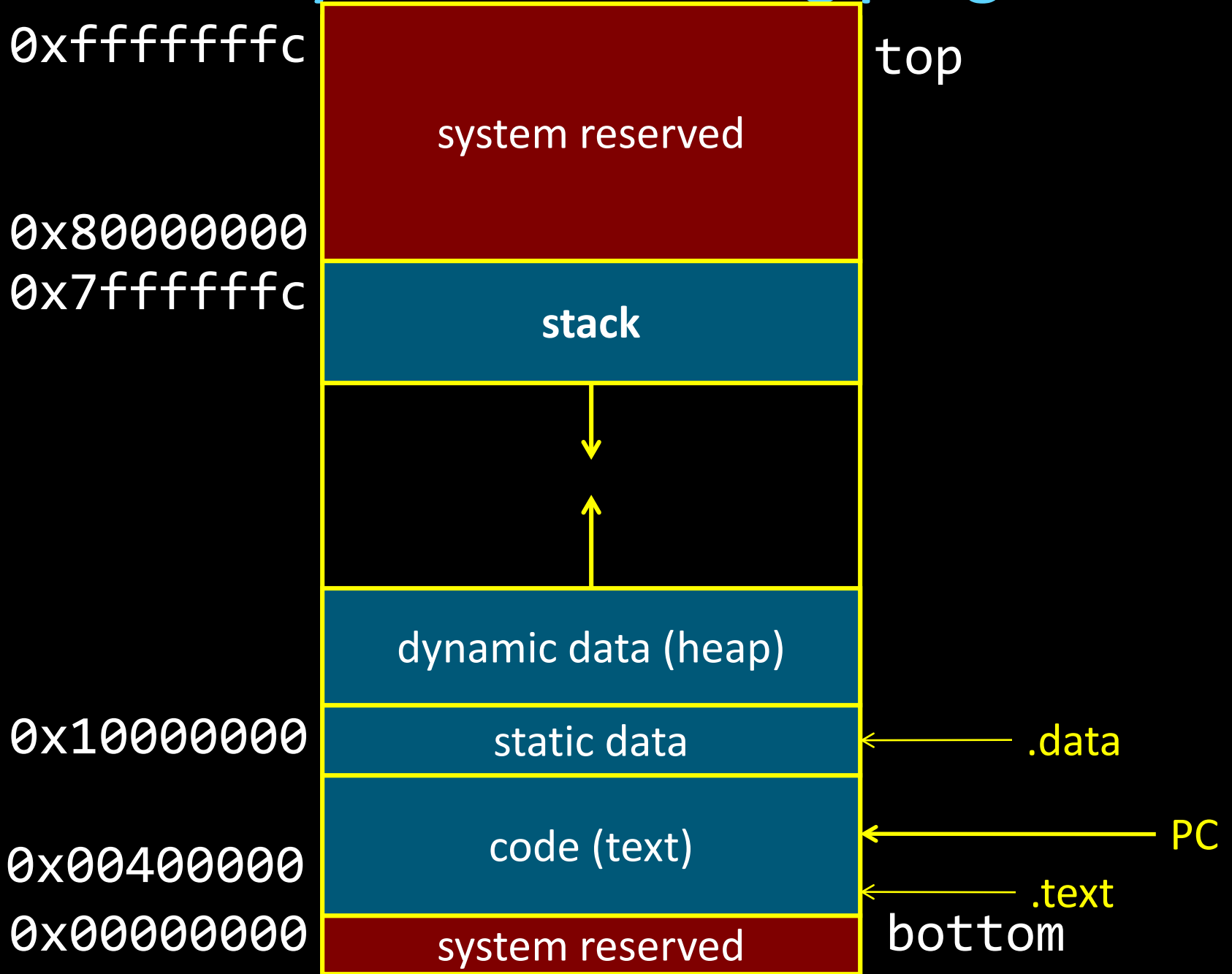
- Exist solely for the duration of the stack frame

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; }`

Anatomy of an executing program



Caller-saved vs. Callee-saved

Callee-save register:

- Assumes register not changed across procedure call
- Thus, **callee must save** the previous contents of the register on procedure entry, restore just before procedure return
- E.g. \$ra, \$fp, \$s0-s7

Caller-save register:

- Assumes that a caller can clobber contents of register
- Thus, **caller must save** the previous contents of the register **before** proc call
- **Caller, then, restores after** the call
- E.g. \$a0-a3, \$v0-\$v1, \$t0-\$t9

MIPS calling convention supports both

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

Save if want to use *after* a call

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

Save *before* use

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-\$t9), 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 subroutine.

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 routine 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 Activation Record (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

Callee-Save

main:

```
addiu $sp,$sp,-32
```

```
sw $31,28($sp)
```

```
sw $30, 24($sp)
```

```
sw $17, 20($sp)
```

```
sw $16, 16($sp)
```

```
addiu $30, $sp, 28
```

...

```
[use $16 and $17]
```

```
lw $31,28($sp)
```

```
lw $30,24($sp)
```

```
lw $17, 20($sp)
```

```
lw $16, 16($sp)
```

```
addiu $sp,$sp,32
```

```
jr $31
```

Assume caller is using the registers

Callee must save on entry, restore on exit

Pays off if caller is actually using the registers, else the save and restore are wasted

Callee-Save

main:

```
addiu $sp,$sp,-32
```

```
sw $ra,28($sp)
```

```
sw $fp, 24($sp)
```

```
sw $s1, 20($sp)
```

```
sw $s0, 16($sp)
```

```
addiu $fp, $sp, 28
```

...

```
[use $s0 and $s1]
```

```
lw $ra,28($sp)
```

```
lw $fp,24($sp)
```

```
lw $s1, 20($sp)
```

```
lw $s0, 16($sp)
```

```
addiu $sp,$sp,32
```

```
jr $ra
```

Assume caller is using the registers

Callee must save on entry, restore on exit

Pays off if caller is actually using the registers, else the save and restore are wasted

Caller-Save

main:

```
...  
[use $8 & $9]  
...  
addiu $sp,$sp,-8  
sw $9, 4($sp)  
sw $8, 0($sp)  
jal mult  
lw $9, 4($sp)  
lw $8, 0($sp)  
addiu $sp,$sp,8  
...  
[use $8 & $9]
```

Assume the registers are free for the taking, clobber them

But since other subroutines will do the same, must protect values that will be used later

By saving and restoring them before and after subroutine invocations

Pays off if a routine makes few calls to other routines with values that need to be preserved

Caller-Save

main:

```
...  
[use $t0 & $t1]  
...  
addiu $sp,$sp,-8  
sw $t1, 4($sp)  
sw $t0, 0($sp)  
jal mult  
lw $t1, 4($sp)  
lw $t0, 0($sp)  
addiu $sp,$sp,8  
...  
[use $t0 & $t1]
```

Assume the registers are free for the taking, clobber them

But since other subroutines will do the same, must protect values that will be used later

By saving and restoring them before and after subroutine invocations

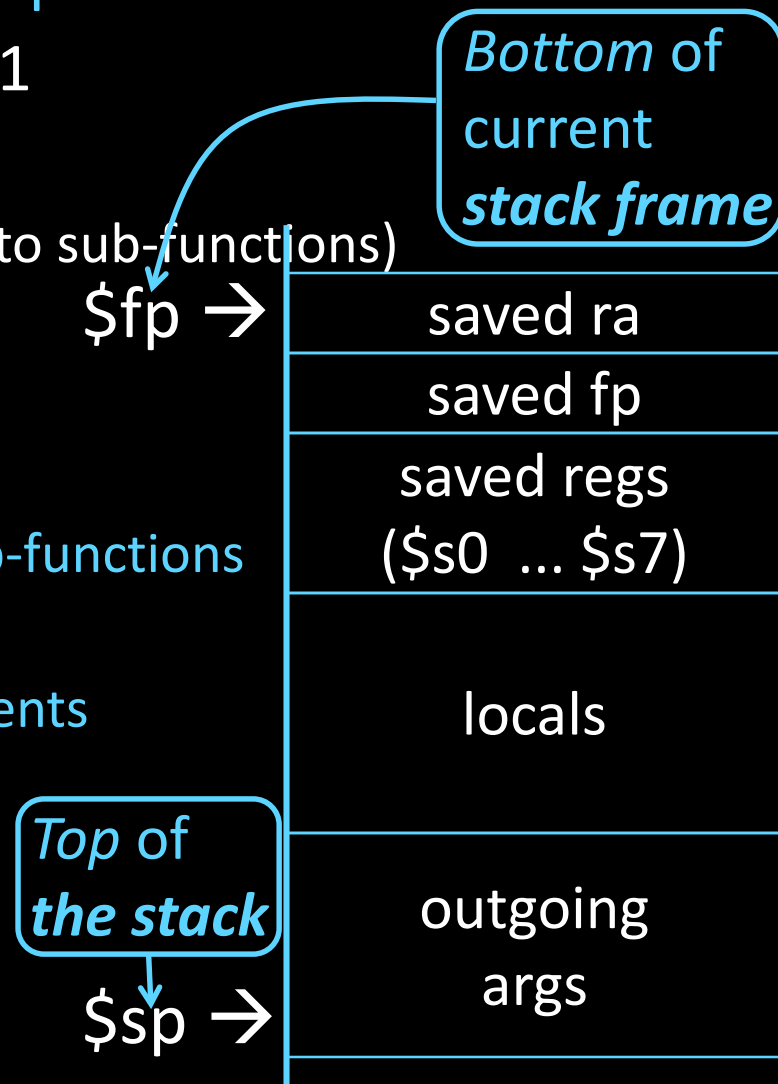
Pays off if a routine makes few calls to other routines with values that need to be preserved

Recap: Conventions so far

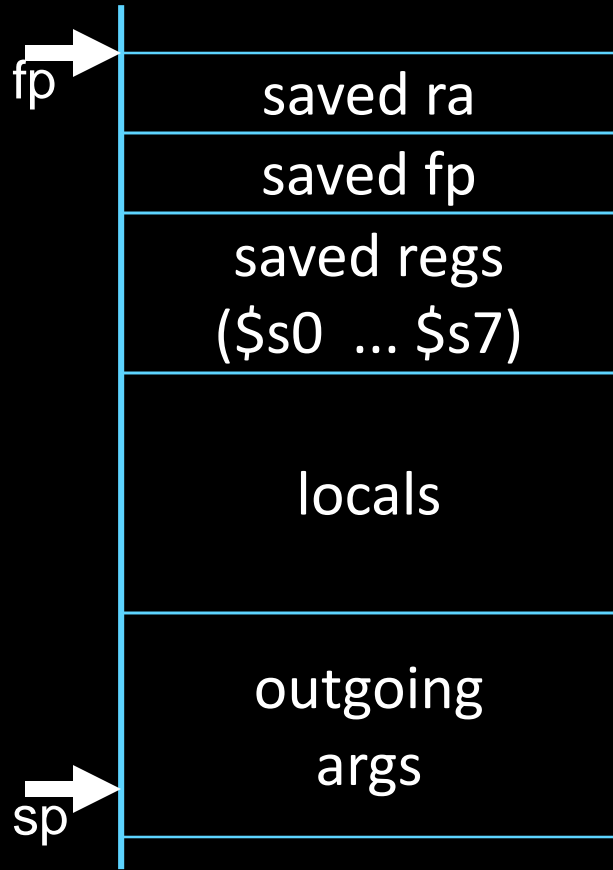
- first four arg words passed in \$a0, \$a1, \$a2, \$a3
- remaining arg words passed in parent's stack frame
- return value (if any) in \$v0, \$v1
- stack frame at \$sp

- 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

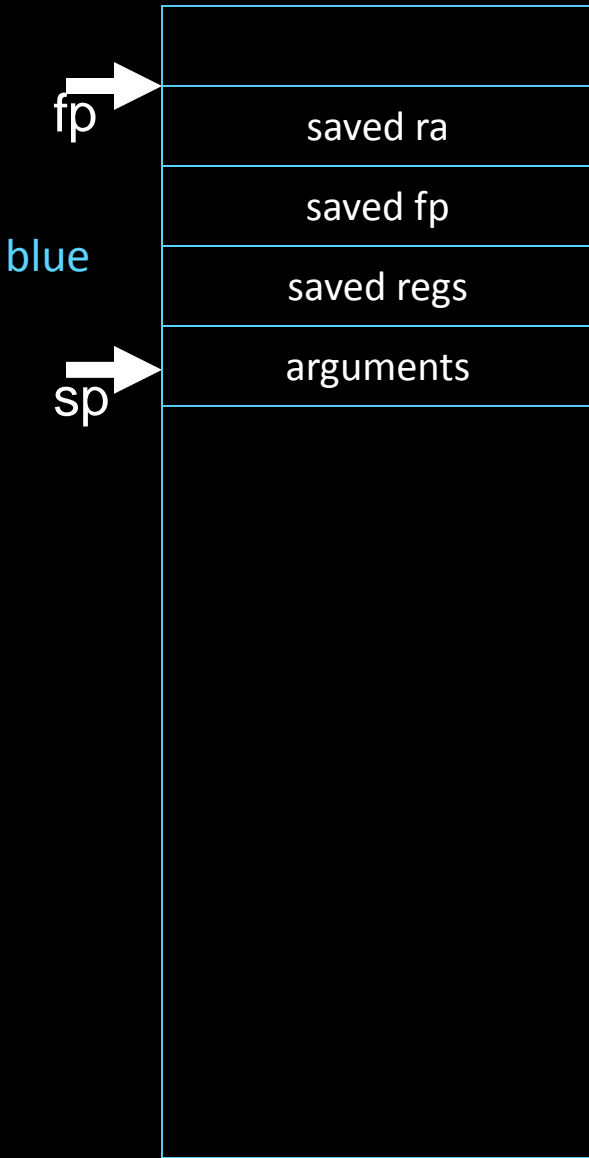


Frame Layout on Stack



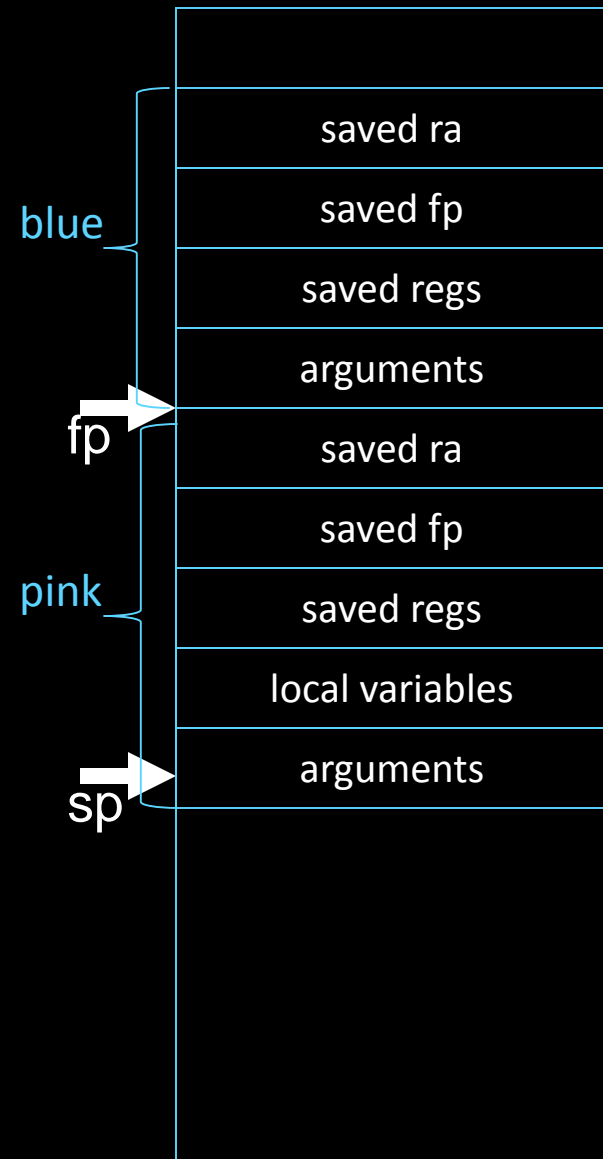
```
ADDIU $sp, $sp, -32 # allocate frame
SW $ra, 28($sp) # save $ra
SW $fp, 24($sp) # save old $fp
SW $s1, 20($sp) # save ...
SW $s0, 16($sp) # save ...
ADDIU $fp, $sp, 28 # set new frame ptr
...
BODY
...
LW $s0, 16($sp) # restore ...
LW $s1, 20($sp) # restore ...
LW $fp, 24($sp) # restore old $fp
LW $ra, 28($sp) # restore $ra
ADDIU $sp, $sp, 32 # dealloc frame
JR $ra
```

Frame Layout on Stack



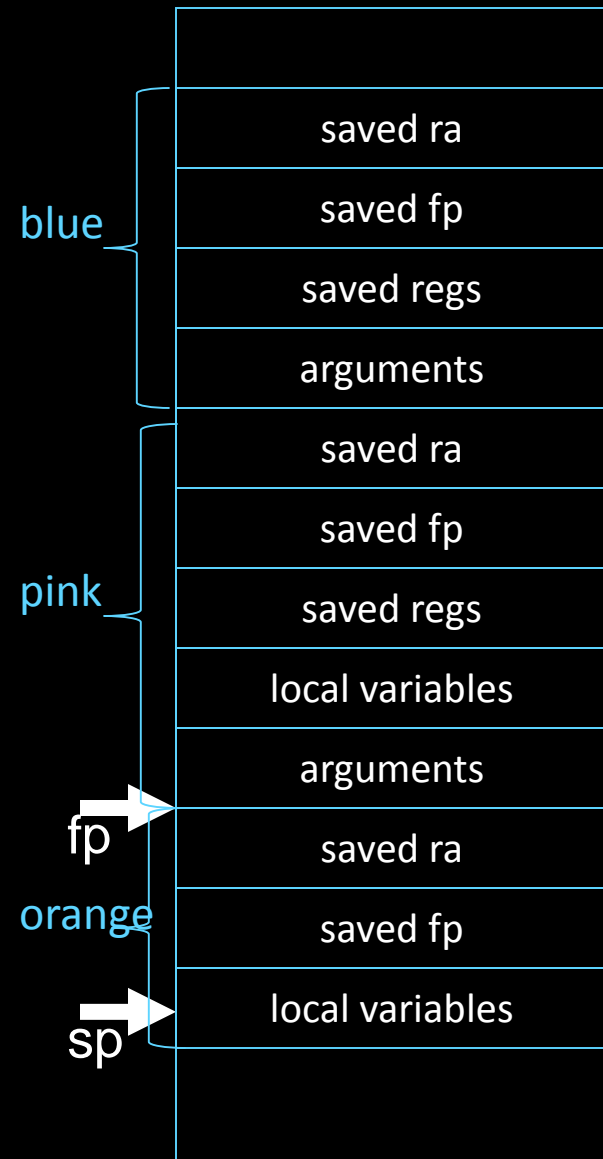
```
blue() {  
    pink(0,1,2,3,4,5);  
}
```

Frame Layout on Stack



```
blue() {  
    pink(0,1,2,3,4,5);  
}  
pink(int a, int b, int c, int d, int e, int f) {  
    orange(10,11,12,13,14);  
}
```

Frame Layout on Stack



```
blue() {  
    pink(0,1,2,3,4,5);  
}  
pink(int a, int b, int c, int d, int e, int f) {  
    orange(10,11,12,13,14);  
}  
orange(int a, int b, int c, int d, int e) {  
    char buf[100];  
    gets(buf); // read string, no check!  
}
```

`buf[100]`

Activity #1: 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;  
}
```

Activity #2: Calling Convention Example:

Prologue, Epilogue

test:

```
# allocate frame
# save $ra
# save old $fp
# callee save ...
# callee save ...
# set new frame ptr
...
...
# restore ...
# restore ...
# restore old $fp
# restore $ra
# dealloc frame
```

Next Goal

Can we optimize the assembly code at all?

Activity #3: 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;  
}
```

How can we optimize
the assembly code?

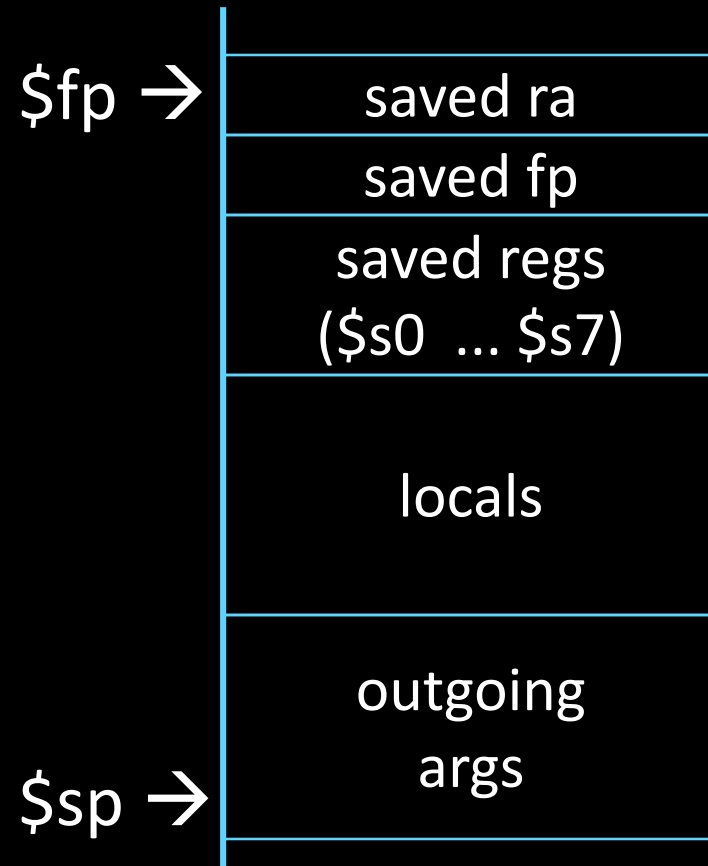
Activity #3: Calling Convention Example: Prologue, Epilogue

test:

```
# allocate frame
# save $ra
# save old $fp
# callee save ...
# callee save ...
# set new frame ptr
...
...
# restore ...
# restore ...
# restore old $fp
# restore $ra
# dealloc frame
```

Minimum stack size for a standard function?

Minimum stack size for a standard function?

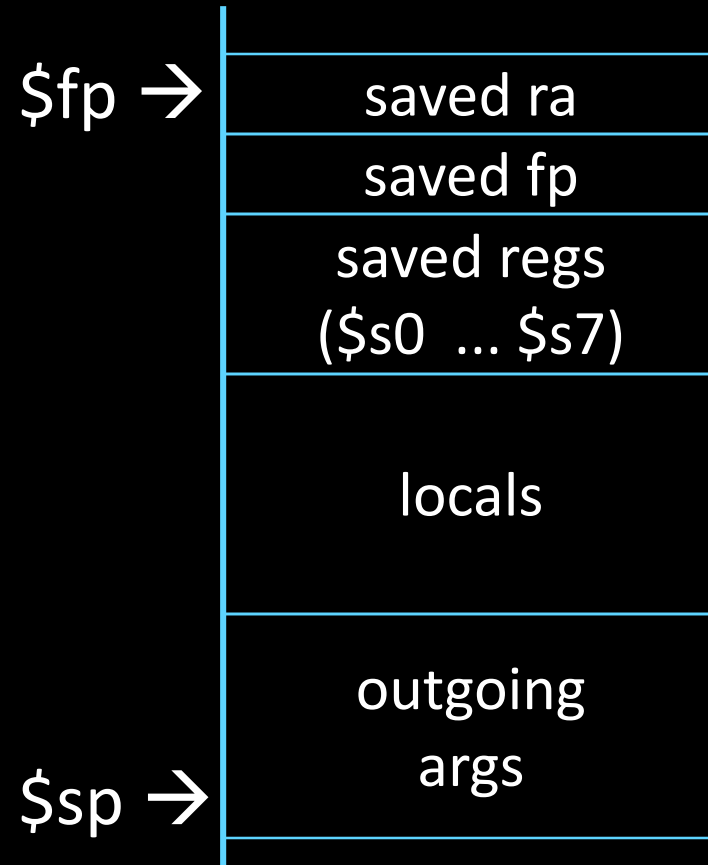


Leaf Functions

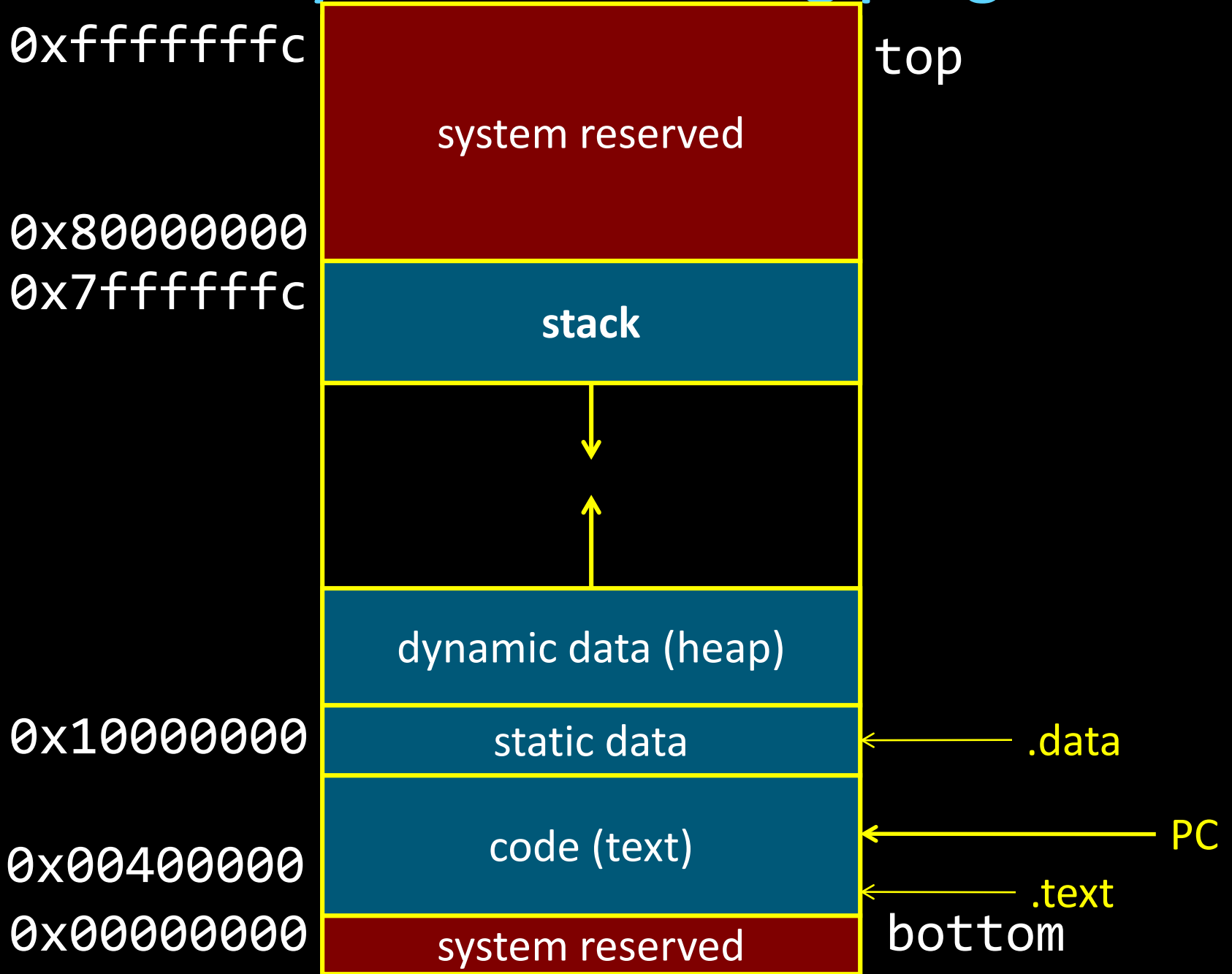
Leaf function does not invoke any other functions

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

Optimizations?



Anatomy of an executing program



Activity #4: Debugging

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

CPU:
\$pc=0x004003C0
\$sp=0x7FFFFFFAC
\$ra=0x00401090

0x00000000

0x0040010c

0x7FFFFFF4

0x00000000

0x00000000

0x00000000

0x00000000

0x004010c4

0x7FFFFFFDC

0x00000000

0x00000000

0x00000015

0x7FFFFFFB0

0x10000004

0x00401090

What func is running?

Who called it?

Has it called anything?

Will it?

Args?

Stack depth?

Call trace?

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

