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 (k0-k1)
Global pointer: $28 (gp)
Assembler temporary: $1 (at)

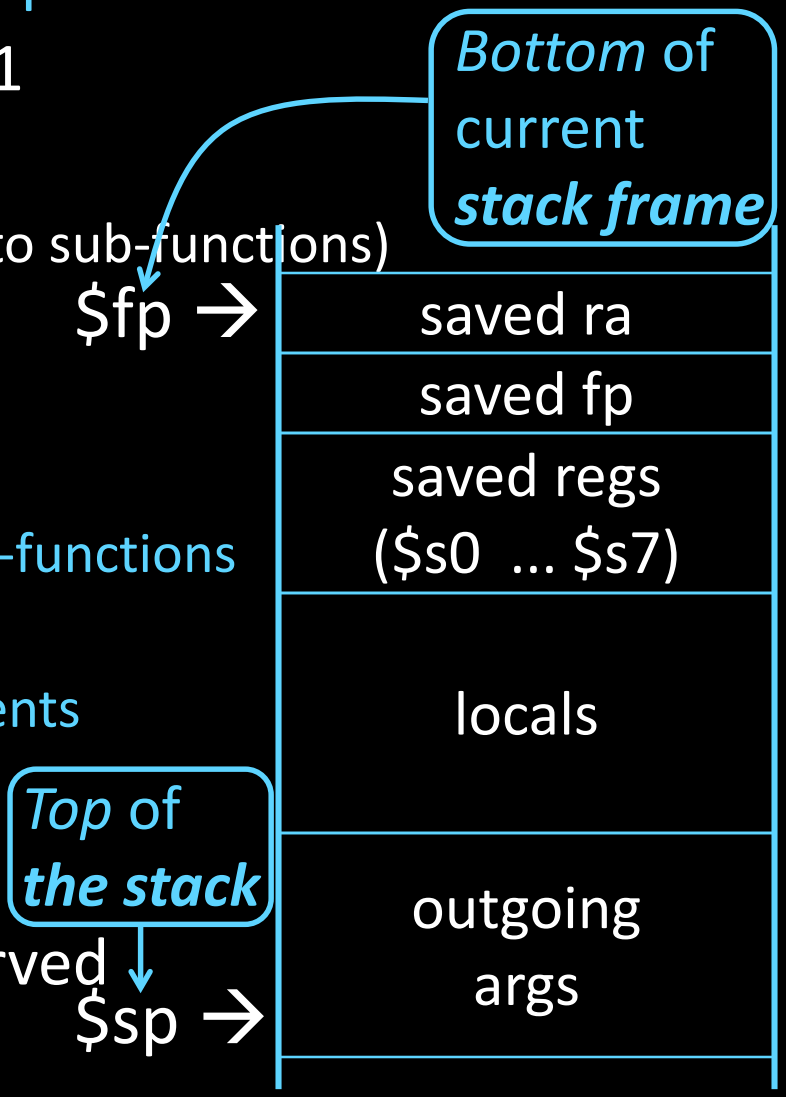
MIPS Register Conventions

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

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 preserved
- 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

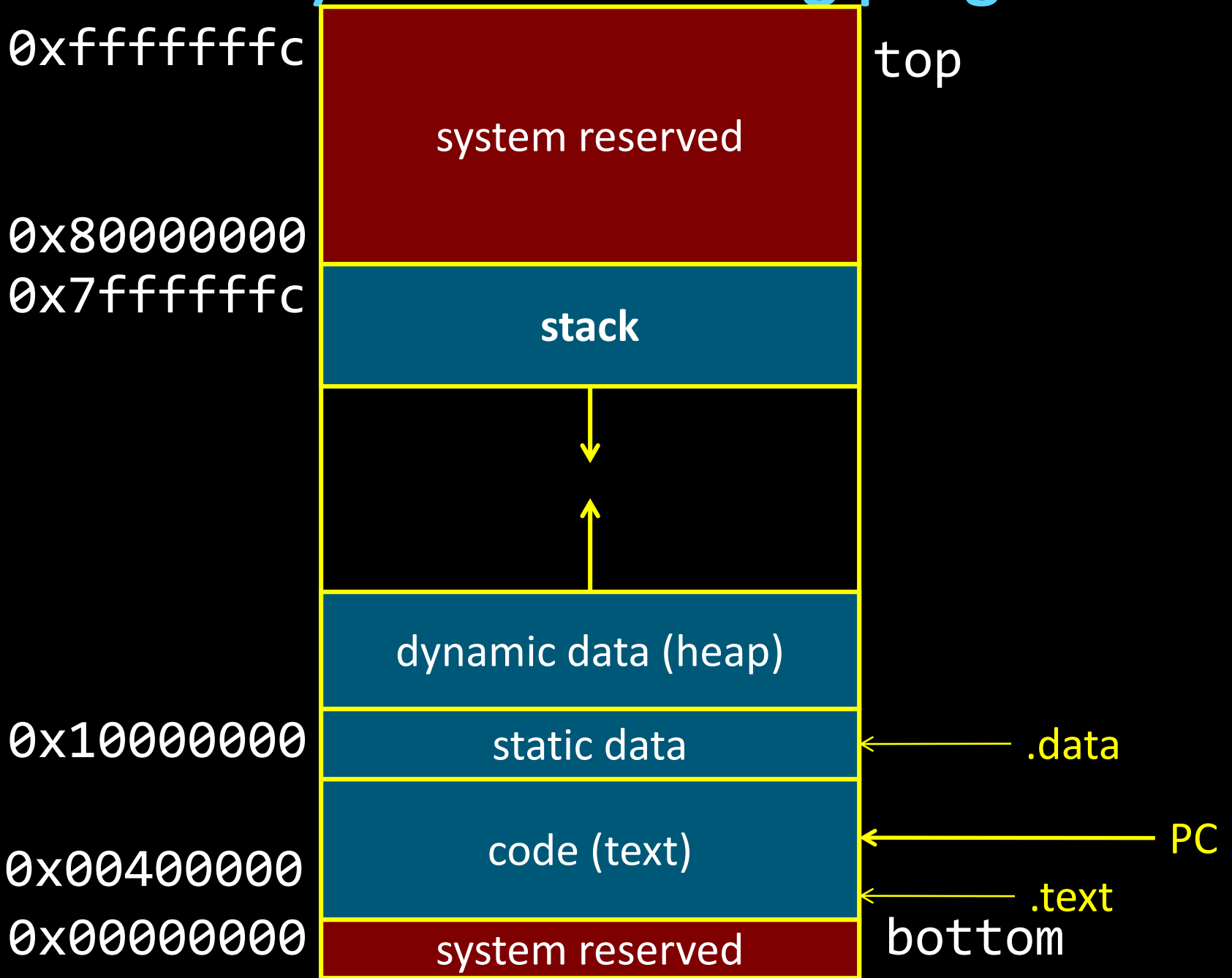
- Exist solely for the duration of the stack frame

Dangling pointers into freed heap mem are bad

Dangling pointers into old stack frames are bad

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

Anatomy of an executing program



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)

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 to subroutine returns
- E.g. $a0-a3, $v0-$v1, $t0-$t9

MIPS calling convention supports both

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, $gp
- Also, $sp

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 to subroutine returns
- 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 does not need to do any extra work at a call site (b/c 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

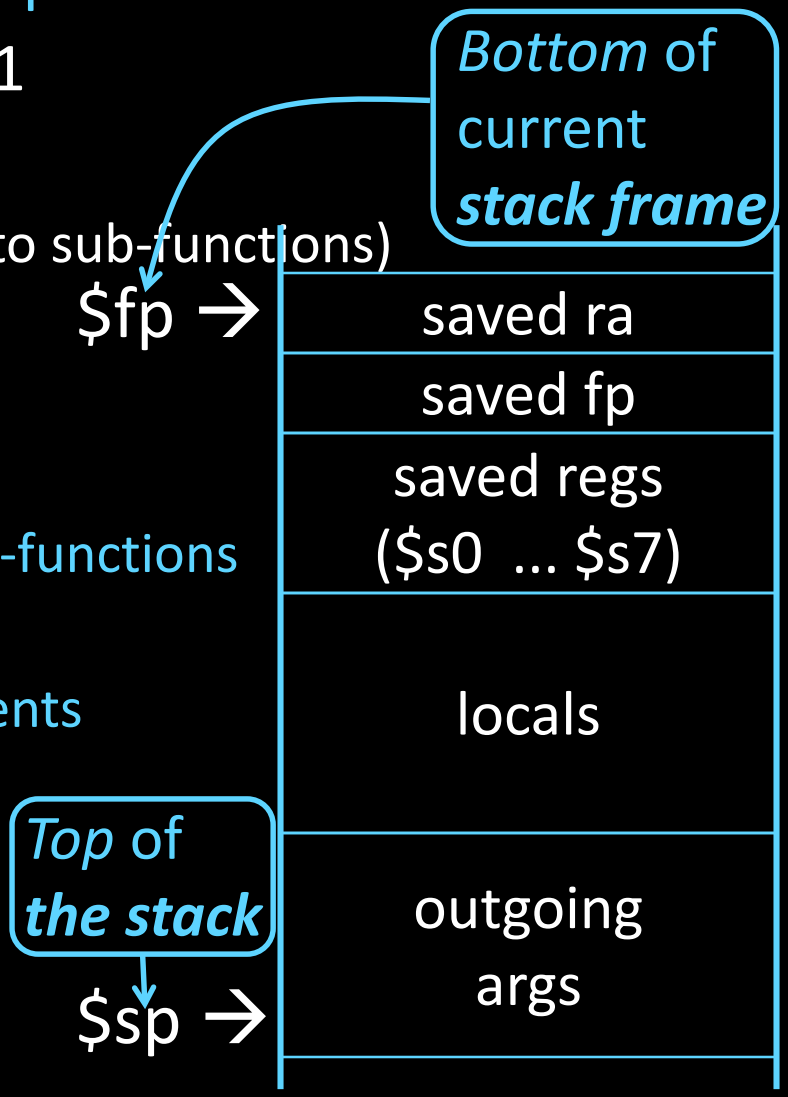
Pays off if a routine makes few calls to other routines with values that need to be preserved

MIPS Register Recap

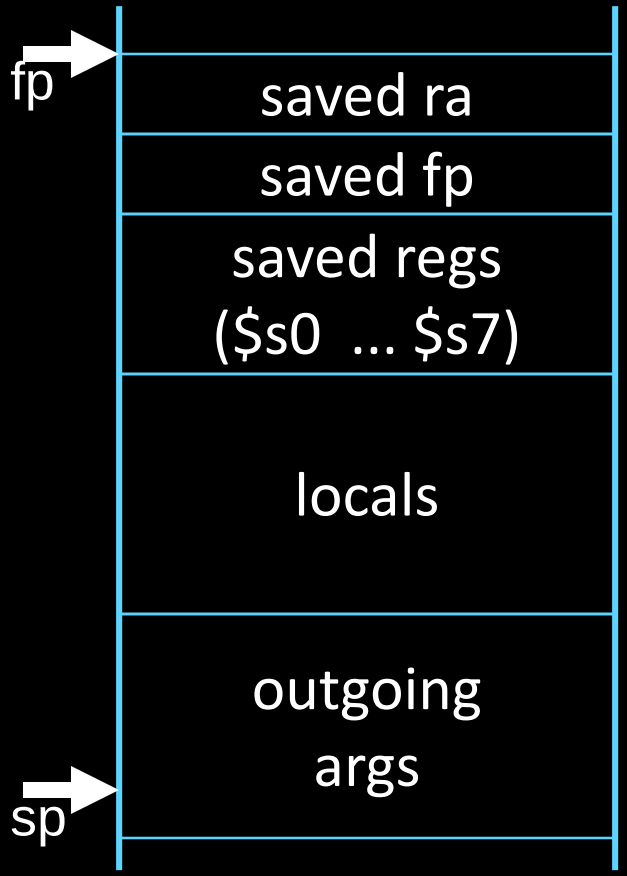
Return address: $31 (ra)
Stack pointer: $29 (sp)
Frame pointer: $30 (fp)
First four arguments: $4-$7 (a0-a3)
Return result: $2-$3 (v0-v1)
Callee-save free regs: $16-$23 (s0-s7)
Caller-save free regs: $8-$15,$24,$25 (t0-t9)
Reserved: $26, $27
Global pointer: $28 (gp)
Assembler temporary: $1 (at)

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



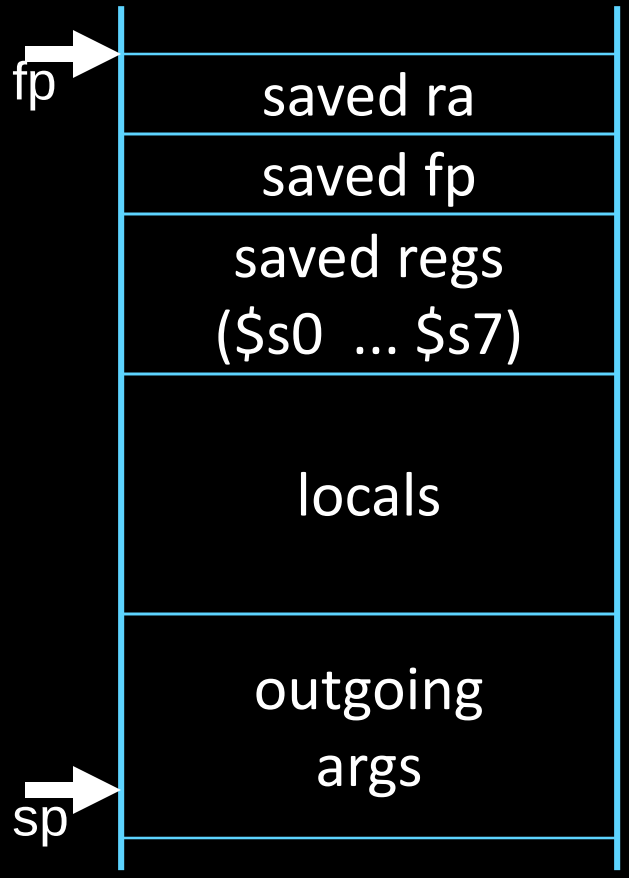
Assume a function uses two callee-save registers.

How do we allocate a stack frame? How large is the stack frame?

What should be stored in the stack frame?

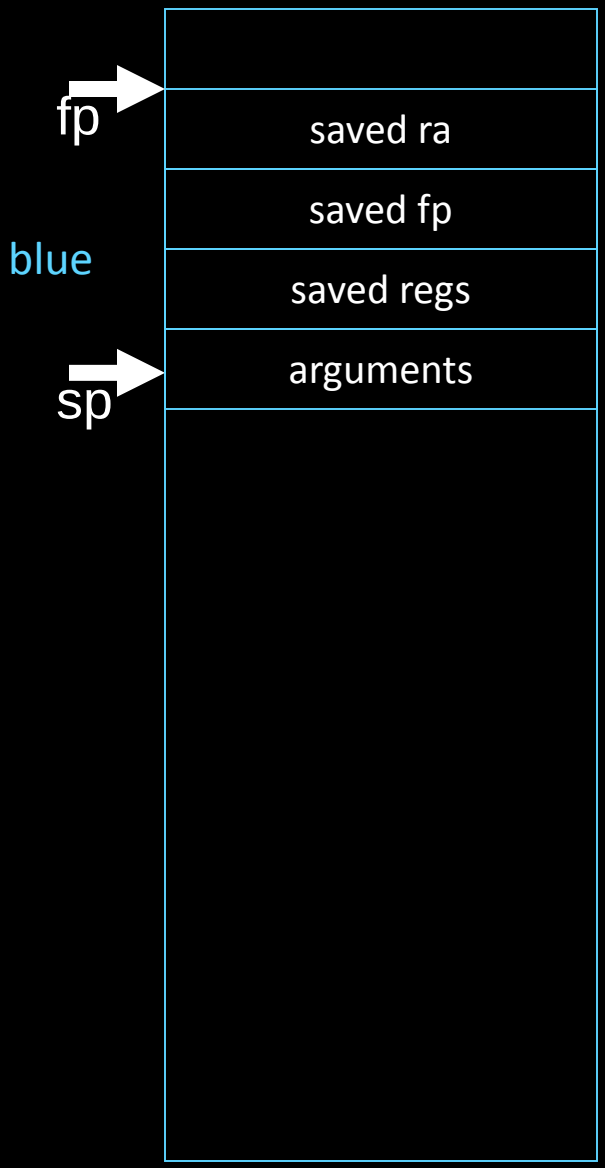
Where should everything be stored?

Frame Layout on Stack



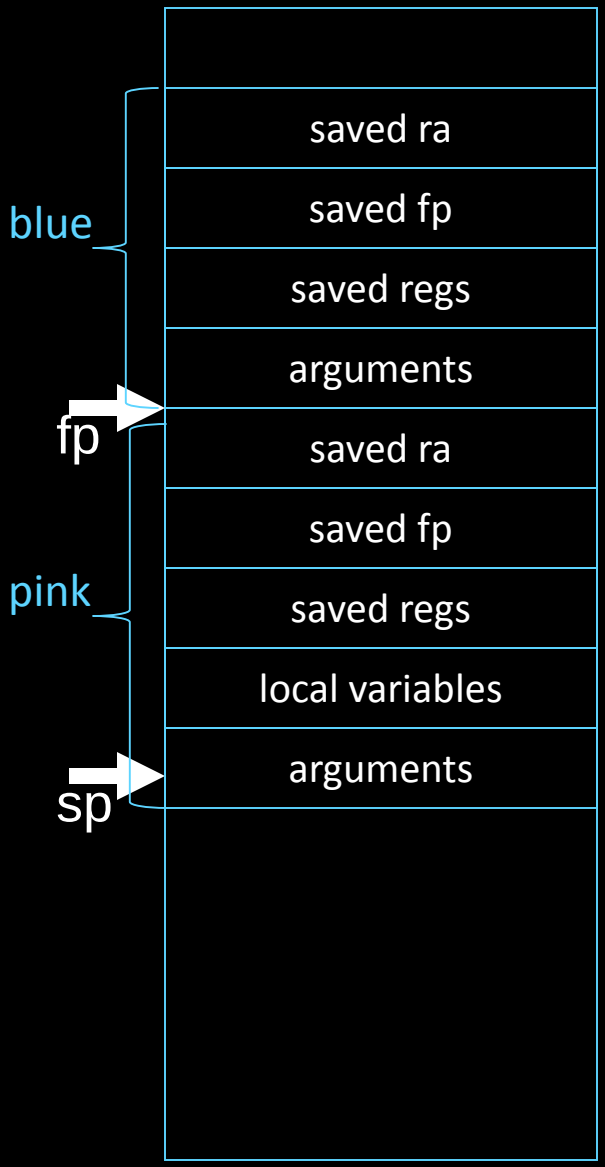

```
ADDIU $sp, $sp, -32   # allocate frame
SW $ra, 28($sp)       # save $ra
SW $fp, 24($sp)       # save old $fp
SW $s1, 20($sp)       # save ...
SW $s0, 16($sp)       # save ...
ADDIU $fp, $sp, 28    # set new frame ptr
…                     ...
BODY
…                     ...
LW $s0, 16($sp)       #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



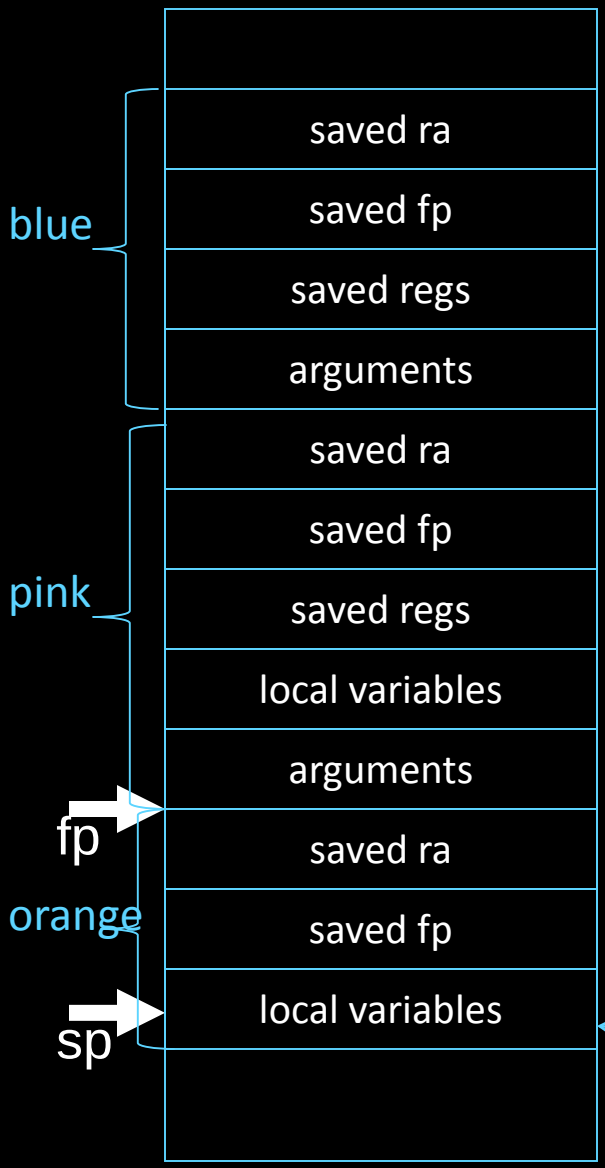

```
blue() {
   pink(0,1,2,3,4,5);
}
```

Frame Layout on Stack



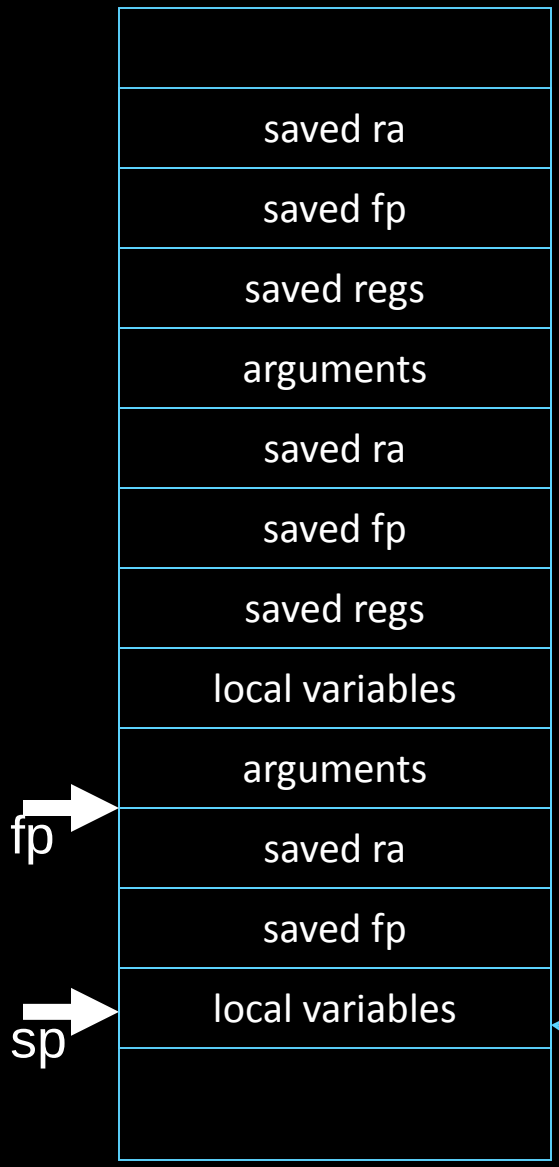

```
blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    orange(10,11,12,13,14);
}
```

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!
}
```

Buffer Overflow




```
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!
}
```

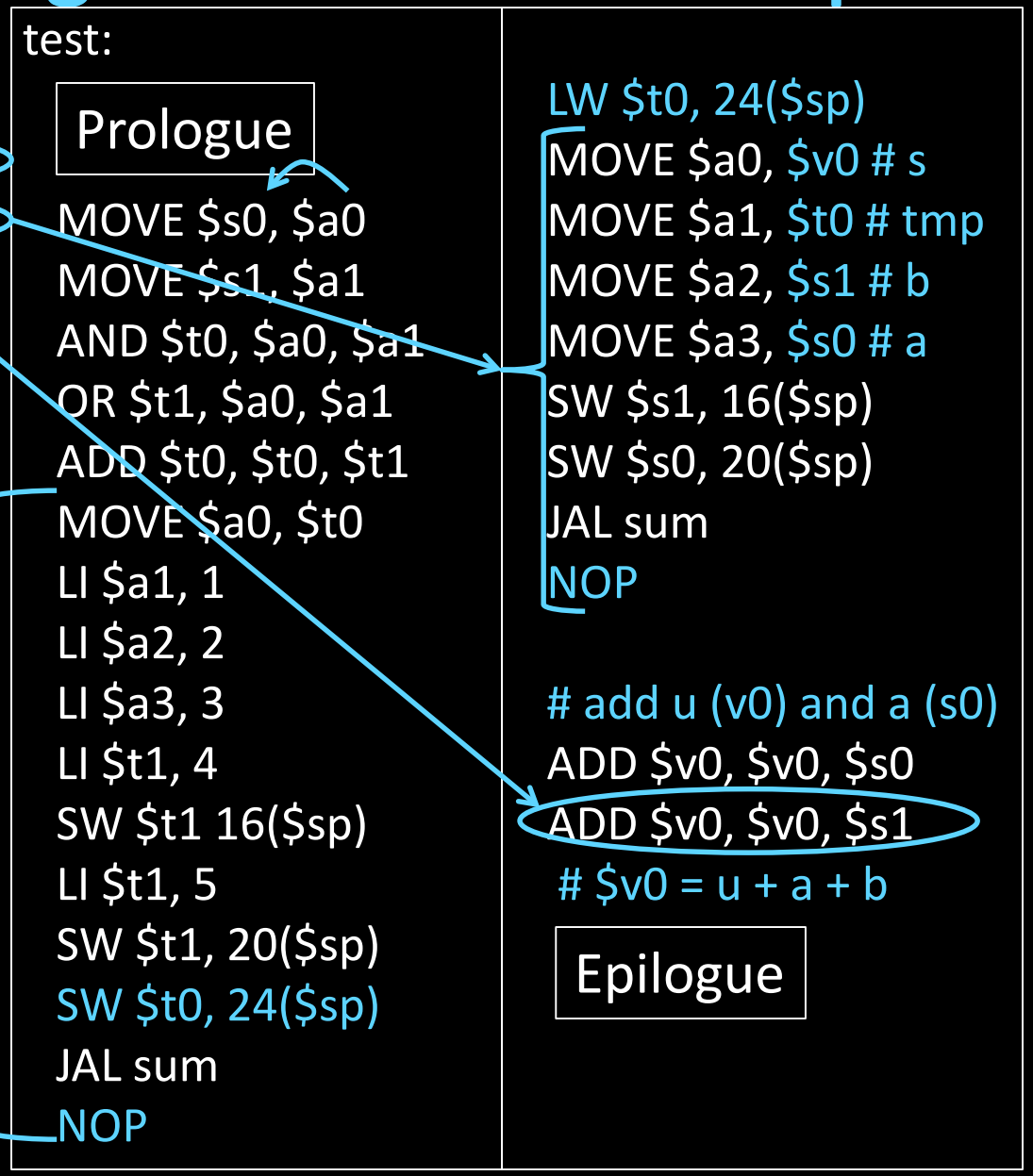
What happens if more than 100 bytes is written to buf?

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 #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 #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;
}
```

How many bytes do we need to allocate for the stack frame?

a) 24
b) 32
c) 40
d) 44
e) 48

```
test:
```

Prologue

```
MOVE $s0, $a0
MOVE $s1, $a1
AND $t0, $a0, $a1
OR $t1, $a0, $a1
ADD $t0, $t0, $t1
MOVE $a0, $t0
LI $a1, 1
LI $a2, 2
LI $a3, 3
LI $t1, 4
SW $t1 16($sp)
LI $t1, 5
SW $t1, 20($sp)
SW $t0, 24($sp)
JAL sum
NOP
```

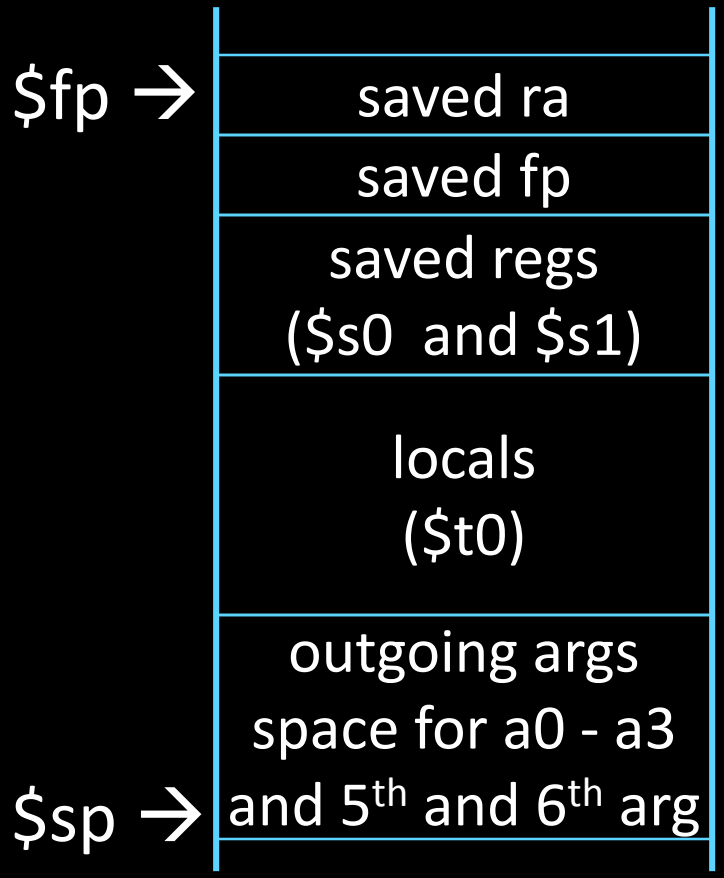
```
LW $t0, 24($sp)
MOVE $a0, $v0 # s
MOVE $a1, $t0 # tmp
MOVE $a2, $s1 # b
MOVE $a3, $s0 # a
SW $s1, 16($sp)
SW $s0, 20($sp)
JAL sum
NOP

# add u (v0) and a (s0)
ADD $v0, $v0, $s0
ADD $v0, $v0, $s1
 # $v0 = u + a + b
```

Epilogue

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;
}
```



test:

Prologue

```
MOVE $s0, $a0
MOVE $s1, $a1
AND $t0, $a0, $a1
OR $t1, $a0, $a1
ADD $t0, $t0, $t1
MOVE $a0, $t0
LI $a1, 1
LI $a2, 2
LI $a3, 3
LI $t1, 4
SW $t1 16($sp)
LI $t1, 5
SW $t1, 20($sp)
SW $t0, 24($sp)
JAL sum
NOP
```

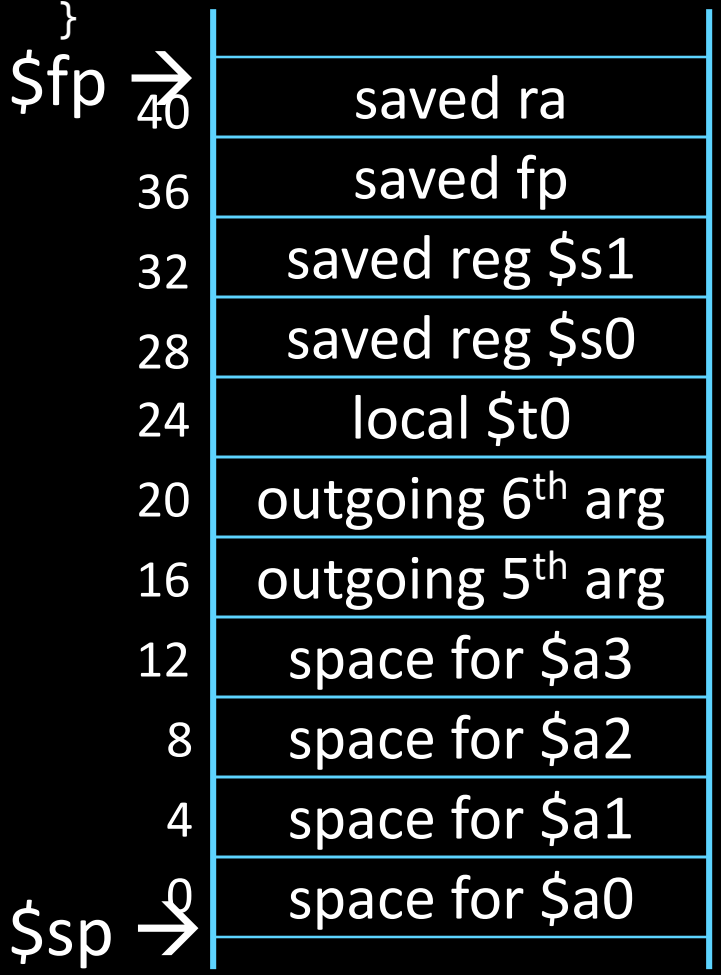
```
LW $t0, 24($sp)
MOVE $a0, $v0 # s
MOVE $a1, $t0 # tmp
MOVE $a2, $s1 # b
MOVE $a3, $s0 # a
SW $s1, 16($sp)
SW $s0, 20($sp)
JAL sum
NOP

# add u (v0) and a (s0)
ADD $v0, $v0, $s0
ADD $v0, $v0, $s1
 # $v0 = u + a + b
```

Epilogue

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;
}
```



test:

Prologue

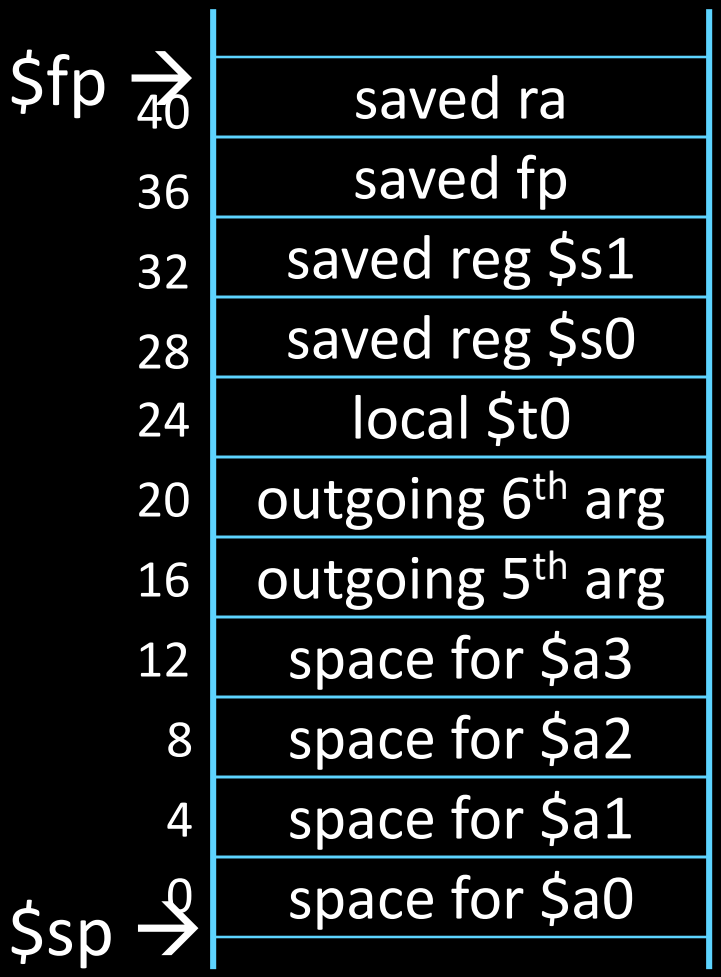
```
MOVE $s0, $a0
MOVE $s1, $a1
AND $t0, $a0, $a1
OR $t1, $a0, $a1
ADD $t0, $t0, $t1
MOVE $a0, $t0
LI $a1, 1
LI $a2, 2
LI $a3, 3
LI $t1, 4
SW $t1 16($sp)
LI $t1, 5
SW $t1, 20($sp)
SW $t0, 24($sp)
JAL sum
NOP
```

```
LW $t0, 24($sp)
MOVE $a0, $v0 # s
MOVE $a1, $t0 # tmp
MOVE $a2, $s1 # b
MOVE $a3, $s0 # a
SW $s1, 16($sp)
SW $s0, 20($sp)
JAL sum
NOP

# add u (v0) and a (s0)
ADD $v0, $v0, $s0
ADD $v0, $v0, $s1
 # $v0 = u + a + b
```

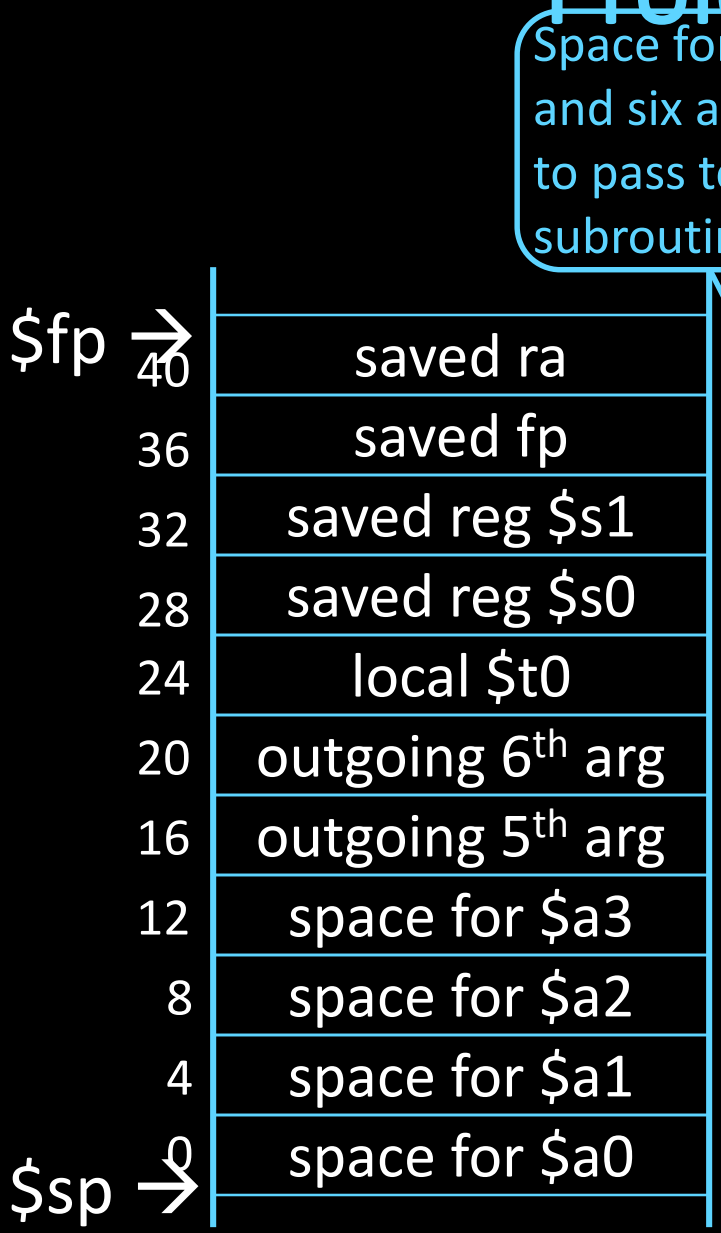
Epilogue

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
```

Activity #2: Calling Convention Example: Prologue, Epilogue



test:

```
ADDIU $sp, $sp, -44     # allocate frame
SW $ra, 40($sp)         # save $ra
SW $fp, 36($sp)         # save old $fp
SW $s1, 32($sp)         # callee save ...
SW $s0, 28($sp)         # callee save ...
ADDIU $fp, $sp, 40      # set new frame ptr
```

Body
(previous slide, Activity #1)

…

…

```
LW $s0, 28($sp)         # restore …
LW $s1, 32($sp)         # restore …
LW $fp, 36($sp)         # restore old $fp
LW $ra, 40($sp)         # restore $ra
ADDIU $sp, $sp, 44      # dealloc frame
JR $ra
NOP
```

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?

test:

Prologue

```
MOVE $s0, $a0
MOVE $s1, $a1
AND $t0, $a0, $a1
OR $t1, $a0, $a1
ADD $t0, $t0, $t1
MOVE $a0, $t0
LI $a1, 1
LI $a2, 2
LI $a3, 3
LI $t1, 4
SW $t1 16($sp)
LI $t1, 5
SW $t1, 20($sp)
SW $t0, 24($sp)
JAL sum
NOP
```

```
LW $t0, 24($sp)
MOVE $a0, $v0 # s
MOVE $a1, $t0 # tmp
MOVE $a2, $s1 # b
MOVE $a3, $s0 # a
SW $s1, 16($sp)
SW $s0, 20($sp)
JAL sum
NOP

# add u (v0) and a (s0)
ADD $v0, $v0, $s0
ADD $v0, $v0, $s1
 # $v0 = u + a + b
```

Epilogue

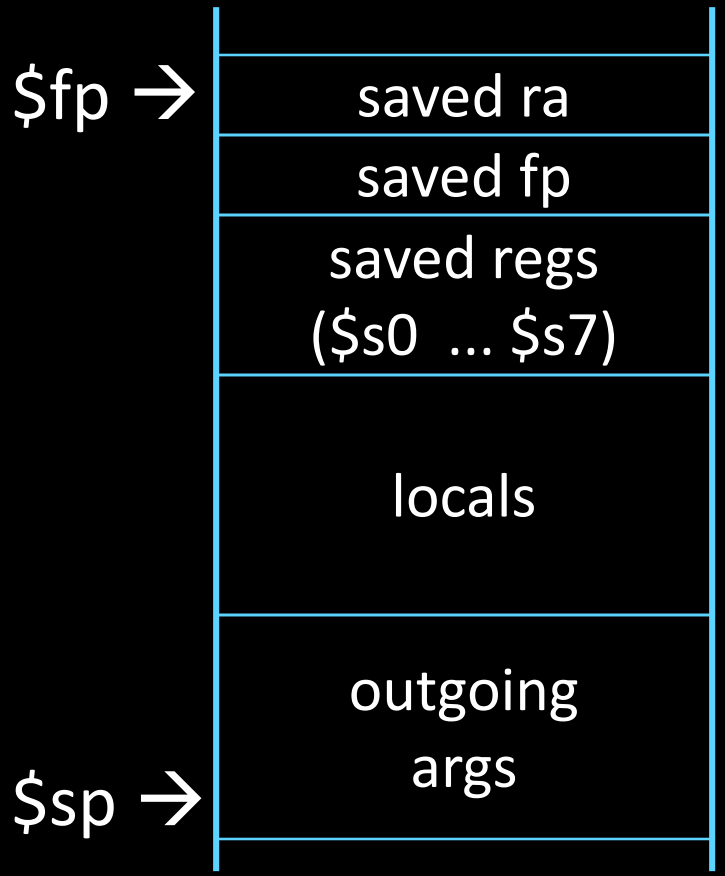
Activity #3: Calling Convention Example: Prologue, Epilogue

```
test:
    ADDIU $sp, $sp, -44      # allocate frame
    SW $ra, 40($sp)          # save $ra
    SW $fp, 36($sp)          # save old $fp
    SW $s1, 32($sp)          # callee save ...
    SW $s0, 28($sp)          # callee save ...
    ADDIU $fp, $sp, 40       # set new frame ptr
        Body                      ...
                                  ...
    LW $s0, 28($sp)          # restore …
    LW $s1, 32($sp)          # restore …
    LW $fp, 36($sp)          # restore old $fp
    LW $ra, 40($sp)          # restore $ra
    ADDIU $sp, $sp, 44       # dealloc frame
    JR $ra
    NOP
```

Minimum stack size for a standard function?

Minimum stack size for a standard function?

24 bytes = 6x 4 bytes ($ra + $fp + 4 args)



Leaf Functions

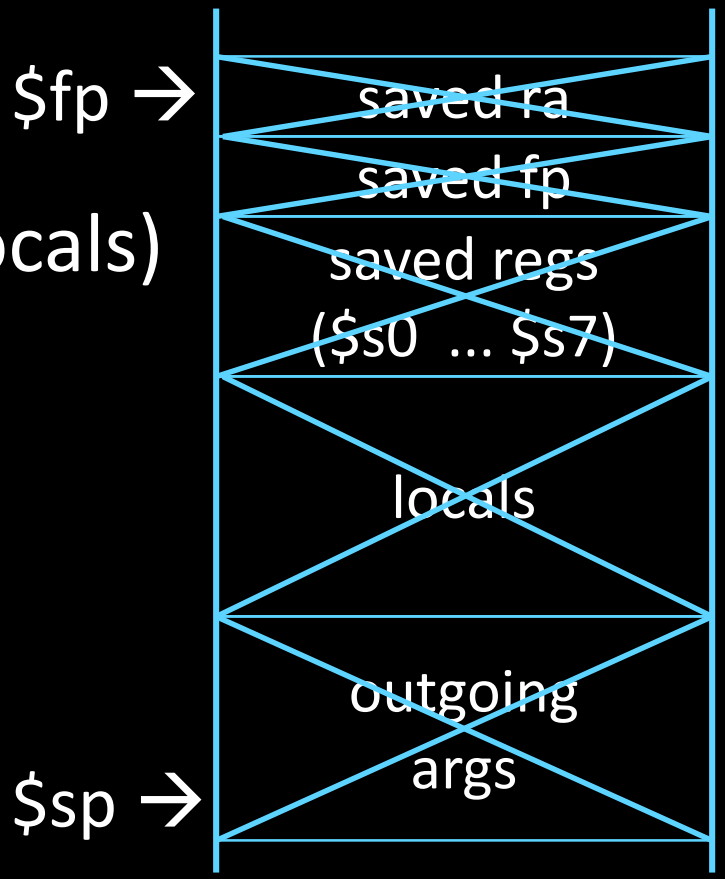
Leaf function does not invoke any other functions

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

Optimizations?

- No saved regs (or locals)
- No outgoing args
- Don't push $ra
- No frame at all?

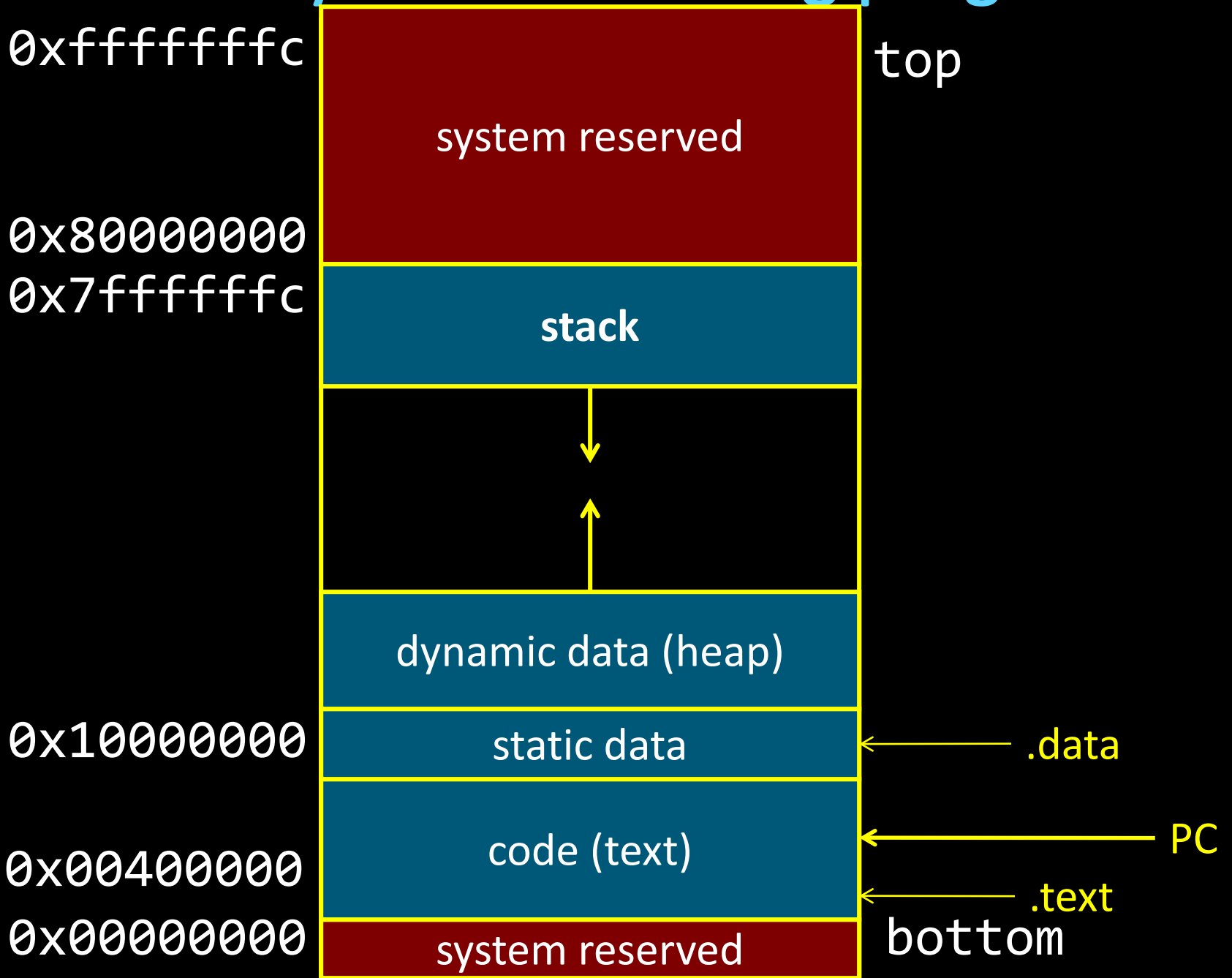
Maybe.



Next Goal

Given a running program (a process), how do we know what is going on (what function is executing, what arguments were passed to where, where is the stack and current stack frame, where is the code and data, etc)?

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=0x7FFFFFAC

$ra=0x00401090

What func is running?

Who called it?

Has it called anything?

Will it?

Args?

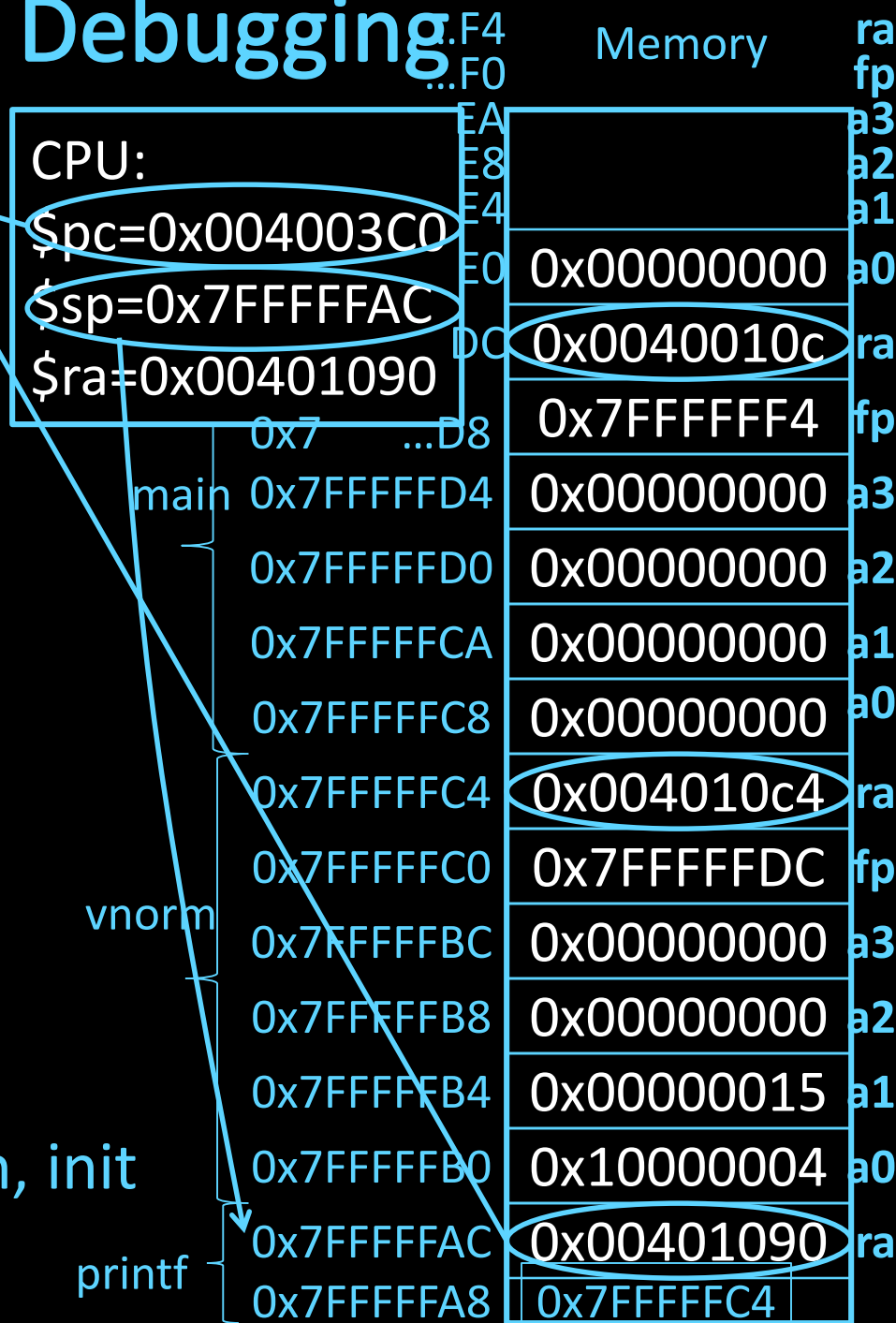
Stack depth?

Call trace?

	0x00000000
	0x0040010c
	0x7FFFFFF4
	0x00000000
	0x00000000
	0x00000000
	0x00000000
	0x004010c4
	0x7FFFFFDC
	0x00000000
	0x00000000
	0x00000015
0x7FFFFFB0	0x10000004
	0x00401090

Activity #4: Debugging

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



What func is running? printf

Who called it? vnorm

Has it called anything? no

Will it? no b/c no space for outgoing args

Args? Str1 and 0x15

Stack depth? 4

Call trace? printf, vnorm, main, init

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