

# Calling Conventions

**Anne Bracy**

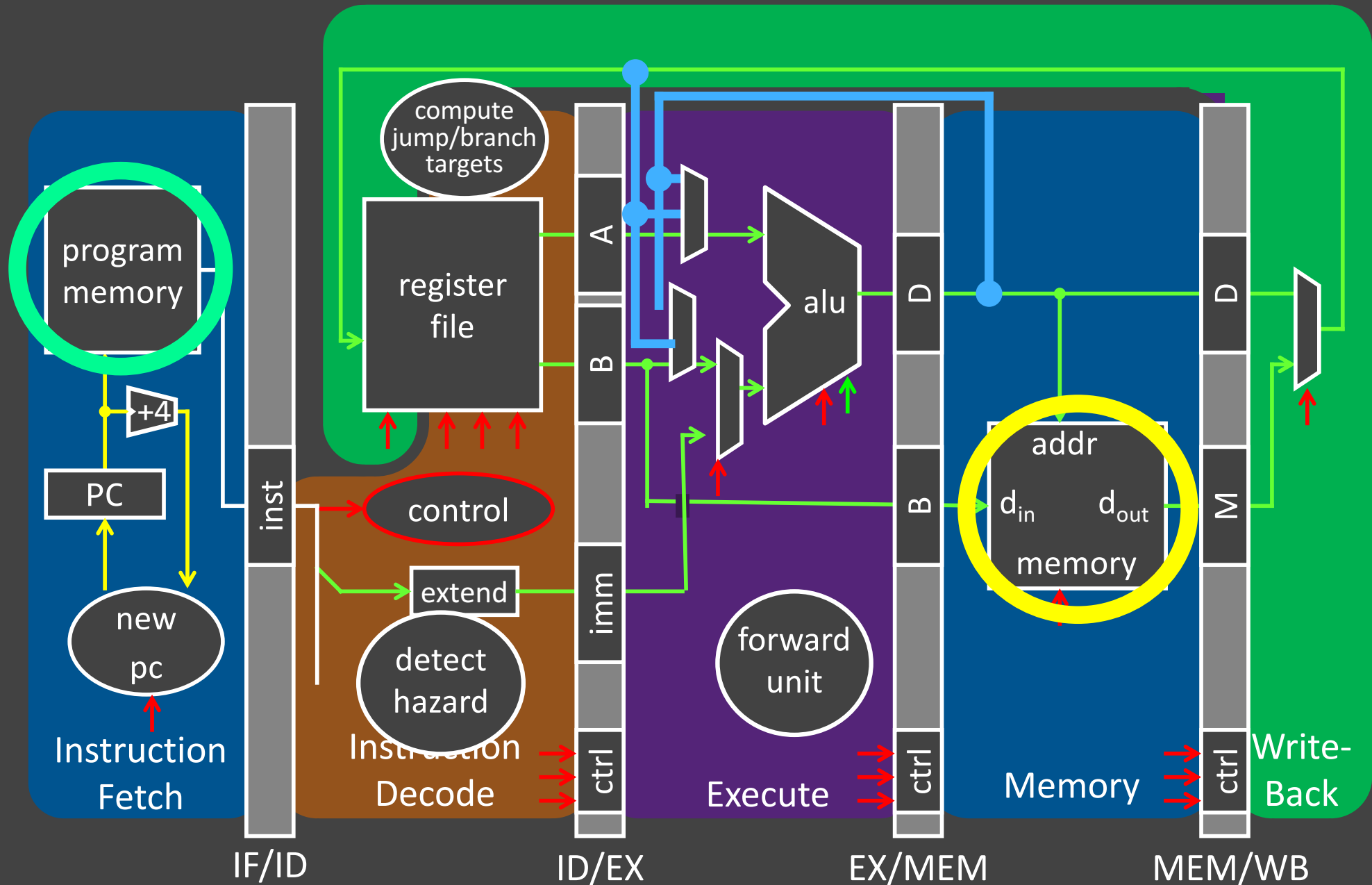
**CS 3410**

Computer Science

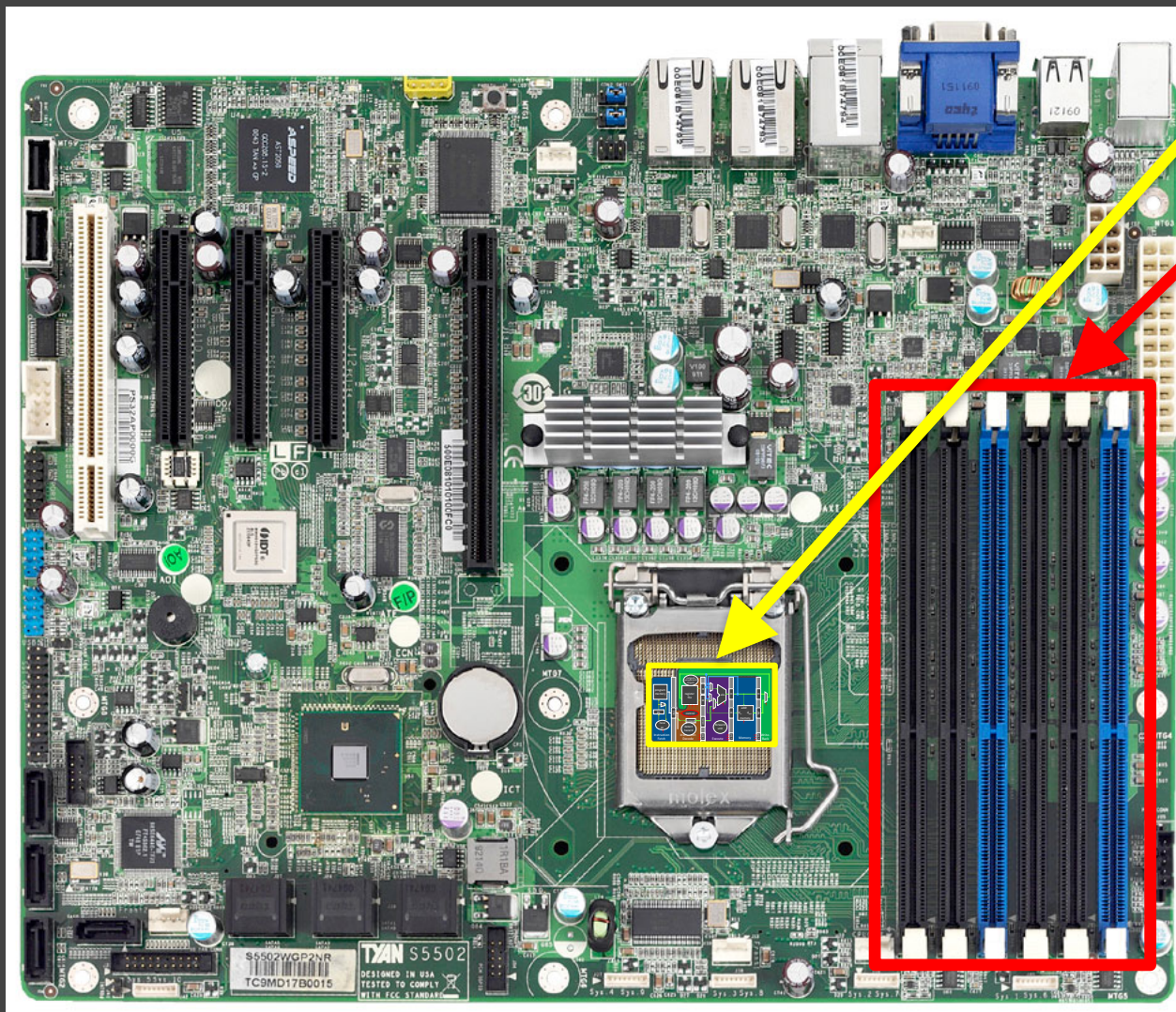
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

# An executing program on chip



# What's the problem?



CPU

Main Memory  
(DRAM)

# An executing program in memory

0xfffffffffc

top

system reserved

0x80000000

0x7fffffff

stack



“Data Memory”

dynamic data (heap)

0x10000000

static data

0x00400000

code (text)

← “Program Memory”

0x00000000

system reserved

bottom

# The Stack

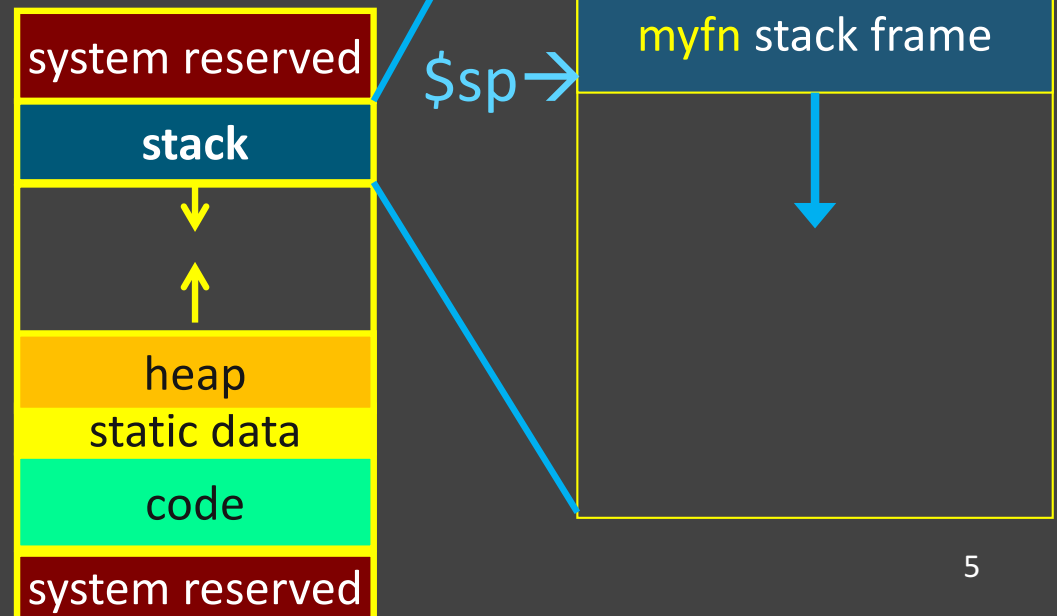
Stack contains stack frames (aka “activation records”)

- 1 stack frame per dynamic function
- Exists only for the duration of function
- Grows down, “top” of stack is `$sp, r29`
- Example: `lw $r1, 0($sp)` puts word at top of stack into `$r1`

Each stack frame contains:

- Local variables, return address (later), register backups (later)

```
int main(...) {  
    ...  
    myfn(x);  
}  
int myfn(int n) {  
    ...  
    myfn();  
}
```

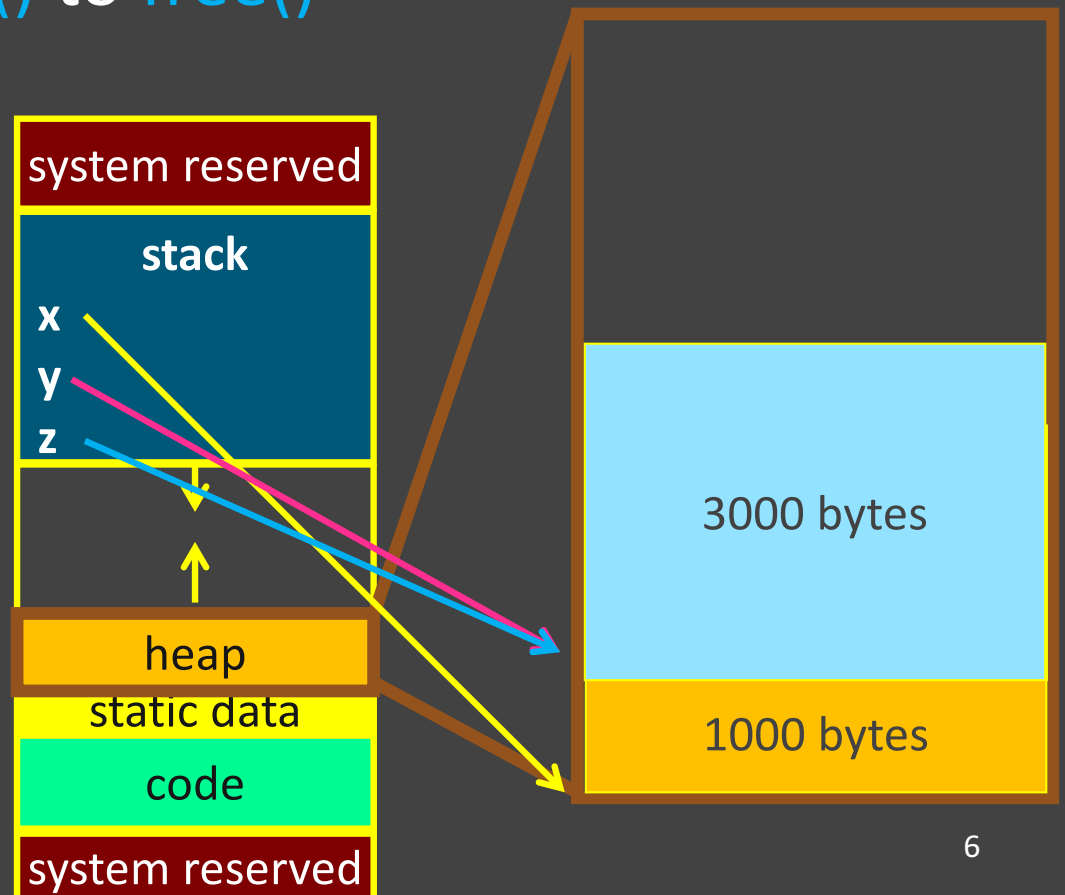


# The Heap

Heap holds dynamically allocated memory

- Program must maintain pointers to anything allocated
  - Example: if \$r3 holds x
    - lw \$r1, 0(\$r3) gets first word x points to
- Data exists from **malloc()** to **free()**

```
void some_function() {  
    int *x = malloc(1000);  
    int *y = malloc(2000);  
    free(y);  
    int *z = malloc(3000);  
}
```

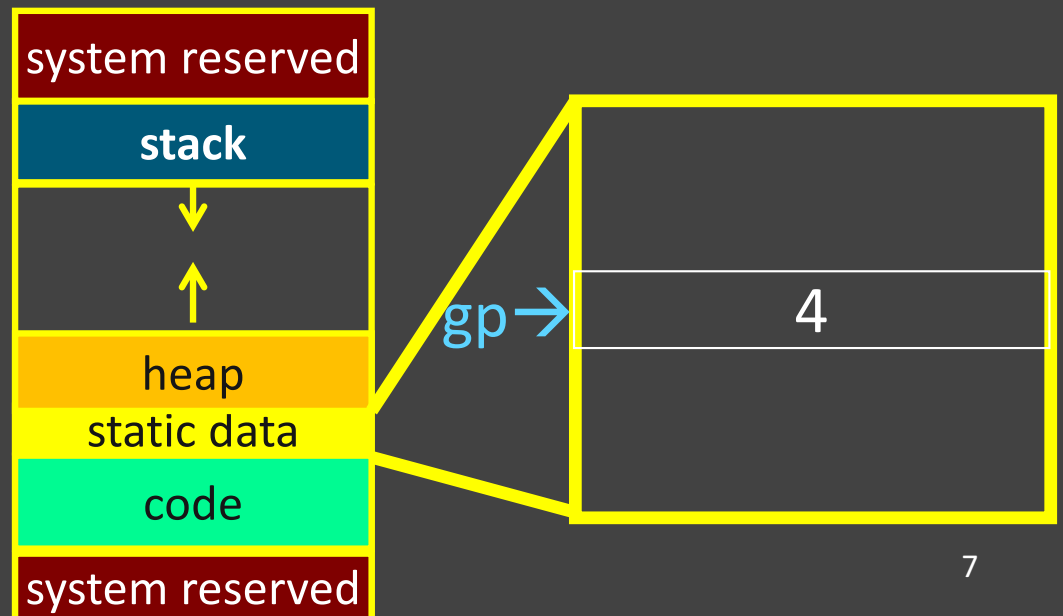


# Data Segment

Data segment contains global variables

- Exist for all time, accessible to all routines
- Accessed w/global pointer
  - `$gp, r28`, points to middle of segment
  - Example: `lw $r1, 0($gp)` gets middle-most word  
(here, `max_players`)

```
int max_players = 4;  
  
int main(...) {  
    ...  
}
```



# Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local			
Global			
Dynamic			

```
int n = 100;
int main (int argc, char* argv[ ]) {
    int i, m = n, sum = 0;
    int* A = malloc(4*m + 4);
    for (i = 1; i <= m; i++) {
        sum += i; A[i] = sum; }
    printf ("Sum 1 to %d is %d\n", n, sum);
}
```

# Globals and Locals

Variables	Visibility	Lifetime	Location
Function-Local i, m, sum, A	w/in function	function invocation	stack
Global n, str	whole program	program execution	.data
Dynamic *A	Anywhere that has a pointer	b/w malloc and free	heap

```
int n = 100;
int main (int argc, char* argv[ ]) {
    int i, m = n, sum = 0;
    int* A = malloc(4*m + 4);
    for (i = 1; i <= m; i++) {
        sum += i; A[i] = sum; }
    printf ("Sum 1 to %d is %d\n", n, sum);
}
```

# Evil things allowed by C

*Don't ever write code like this!*

## Dangling pointers into freed heap mem

```
void some_function() {  
    int *x = malloc(1000);  
    int *y = malloc(2000);  
    free(y);  
    int *z = malloc(3000);  
    y[20] = 7;  
}
```

## Dangling pointers into old stack frames

```
void f1() {  
    int *x = f2();  
    int y = *x + 2;  
}  
int *f2() {  
    int a = 3;  
    return &a;  
}
```

# iClicker Question

Which of the following is trouble-free code?

**A**

```
int *bubble()
{ int a;
  ...
  return &a;
}
```

**B**

```
char *rubble()
{ char s[20];
  gets(s);
  return s;
}
```

**C**

```
int *toil()
{ s = malloc(20);
  ...
  return s;
}
```

**D**

```
int *trouble()
{ s = malloc(20);
  ...
  free(s);
  ...
  return s;
}
```

# How does a function call work?

```
int main (int argc, char* argv[ ]) {  
    int n = 9;  
    int result = myfn(n);  
}
```

```
int myfn(int n) {  
    int f = 1;  
    int i = 1;  
    int j = n - 1;  
    while(j >= 0) {  
        f *= i;  
        i++;  
        j = n - i;  
    }  
    return f;  
}
```

# Calling Convention for Procedure Calls

## Transfer Control

- Caller → Routine
- Routine → Caller

## Pass Arguments to and from the routine

- fixed length, variable length, recursively
- Get return value back to the caller

## Manage Registers

- Allow each routine to use registers
- Prevent routines from clobbering each others' data

### **What is a Convention?**

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

# Jumps are not enough



Jumps to the callee

Jumps back

What about multiple sites?

# Jump-and-Link / Jump Register

*First call*

r31 *after1*



JAL saves the PC in register \$31

Subroutine returns by jumping to \$31

# Jump-and-Link / Jump Register

*Second call*

r31 *after2*



JAL saves the PC in register \$31

Subroutine returns by jumping to \$31

What happens for recursive invocations?

# JAL / JR for Recursion?

*First call*

r31 **after1**

```
main:
    jal myfn
after1:
    add $1,$2,$3
```

1

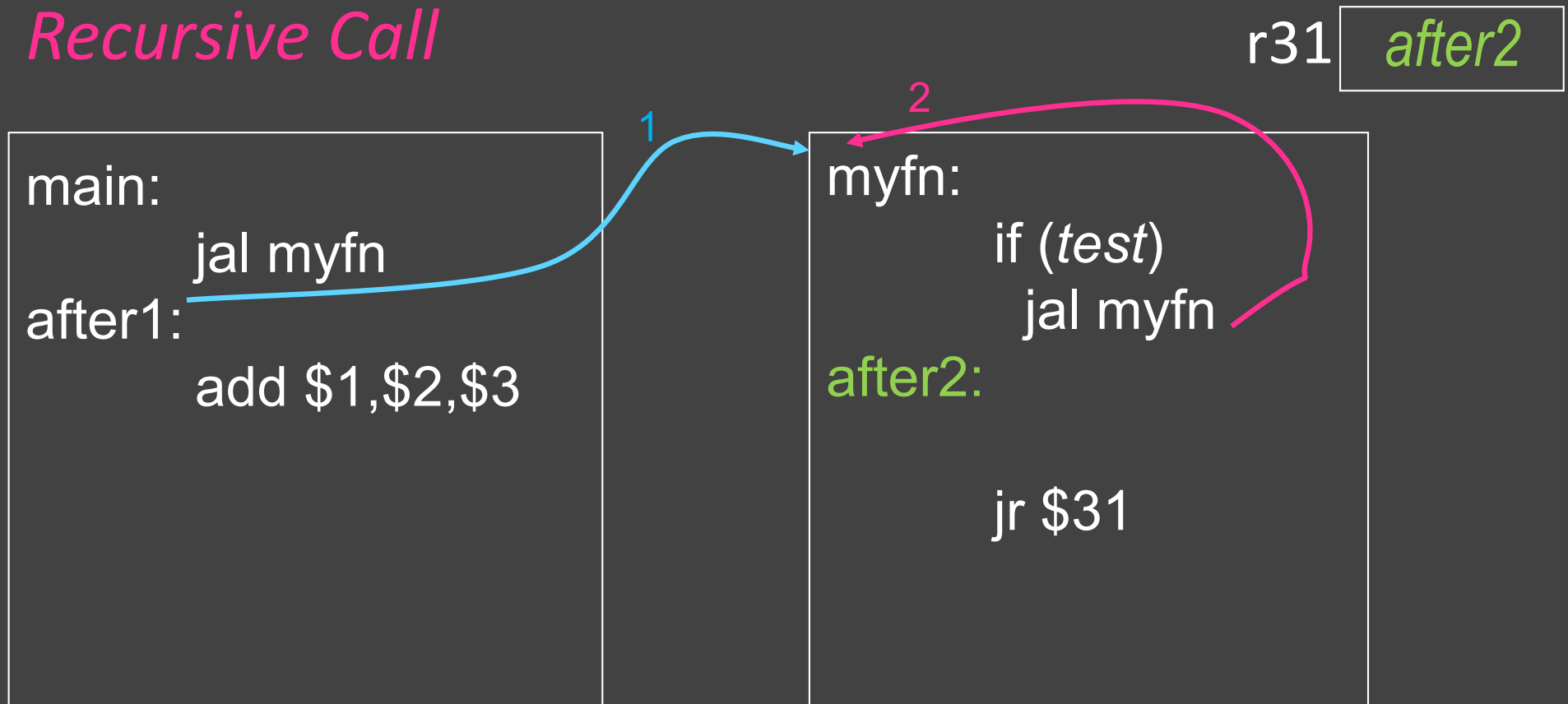
```
myfn:
    if (test)
        jal myfn
    after2:
        jr $31
```

Problems with recursion:

- overwrites contents of \$31

# JAL / JR for Recursion?

## Recursive Call

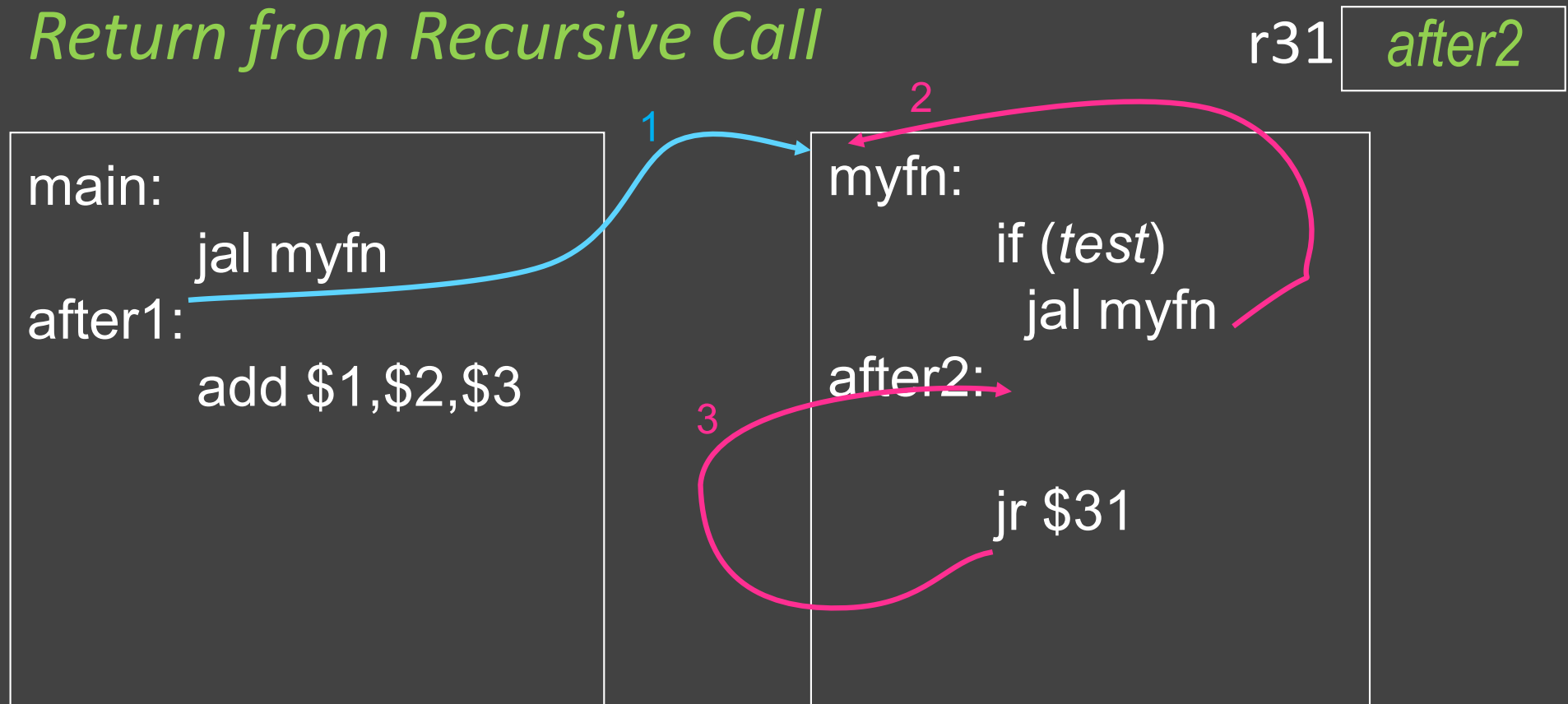


## Problems with recursion:

- overwrites contents of \$31

# JAL / JR for Recursion?

## *Return from Recursive Call*

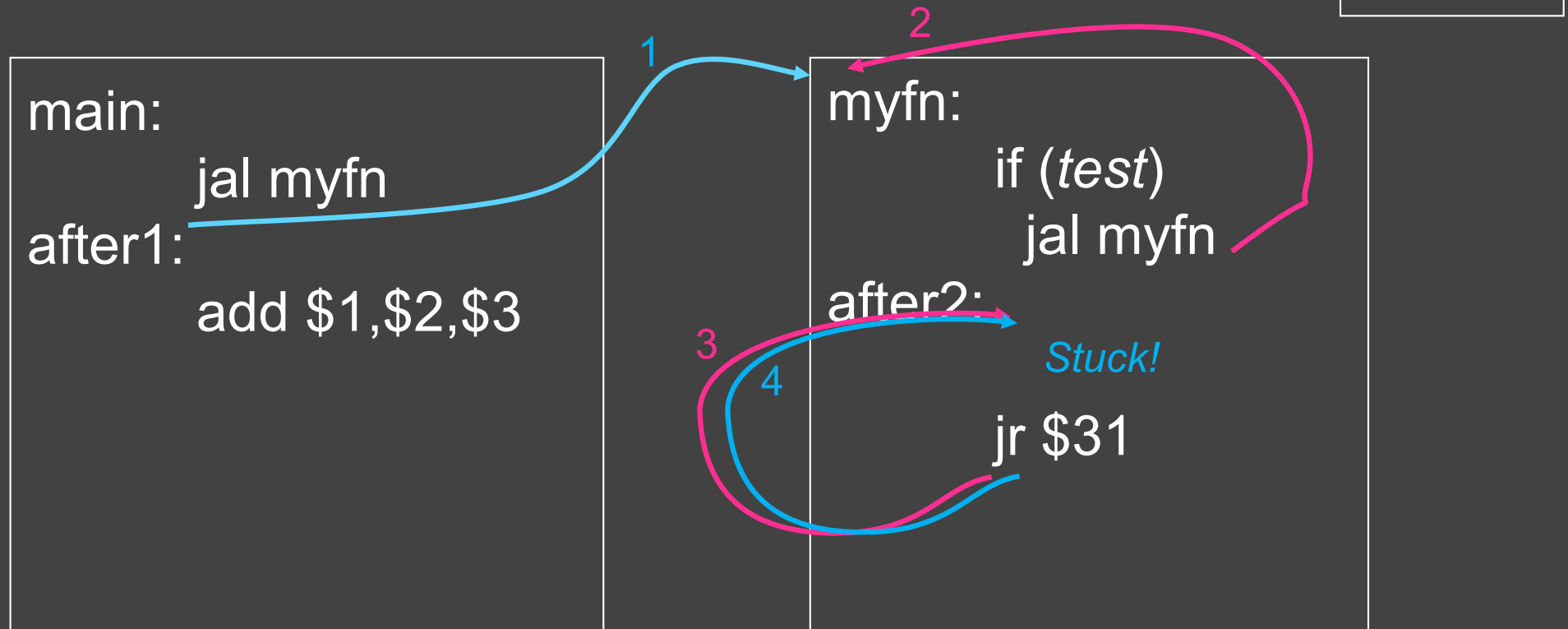


## Problems with recursion:

- overwrites contents of \$31

# JAL / JR for Recursion?

*Return from Original Call ???*



Problems with recursion:

- overwrites contents of \$31

# Return Address lives in Stack Frame

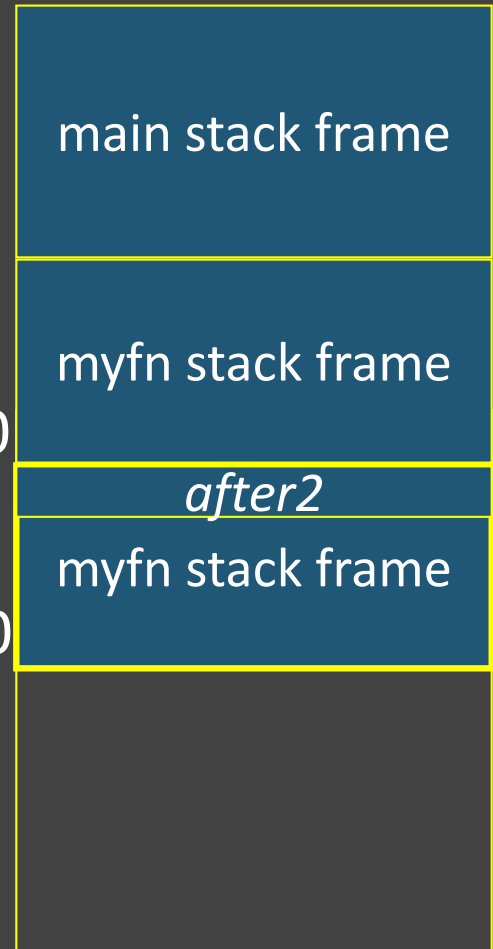
Stack Manipulated by push/pop operations

*Context: after 2<sup>nd</sup> JAL to myfn (from myfn)*

PUSH: ADDIU \$sp, \$sp, -20 // move sp down  
SW \$31, 16(\$sp) // store retn PC 1<sup>st</sup>

*Context: 2<sup>nd</sup> myfn is done (r31 == ???)*

POP: LW \$31, 16(\$sp) // restore retn PC → r31  
ADDIU \$sp, \$sp, 20 // move sp up  
JR \$31 // return



r29 x1FD0

r31 XXXX

*For now: Assume each frame = x20 bytes  
(just to make this example concrete)*

# iClicker Question

Why do we need a JAL instruction for procedure calls?

- A. The only way to change the PC of your program is with a JAL instruction.
- B. The system won't let you jump to a procedure with just a JMP instruction.
- C. If you JMP to a function, it doesn't know where to return to upon completion.
- D. Actually, JAL only works for the first function call. With multiple active functions, JAL is not the right instruction to use.

# Calling Convention for Procedure Calls

## ~~Transfer Control~~

- ~~• Caller → Routine~~
- ~~• Routine → Caller~~

## Pass Arguments to and from the routine

- fixed length, variable length, recursively
- Get return value back to the caller

## Manage Registers

- Allow each routine to use registers
- Prevent routines from clobbering each others' data

# Simple Argument Passing (1-4 args)

```
main() {  
    int x = myfn(6, 7);  
    x = x + 2;  
}
```

```
main:  
    li $a0, 6  
    li $a1, 7  
    jal myfn  
    addi $r1, $v0, 2
```

First four arguments:

passed in registers \$4-\$7

- aka \$a0, \$a1, \$a2, \$a3

Returned result:

passed back in a register

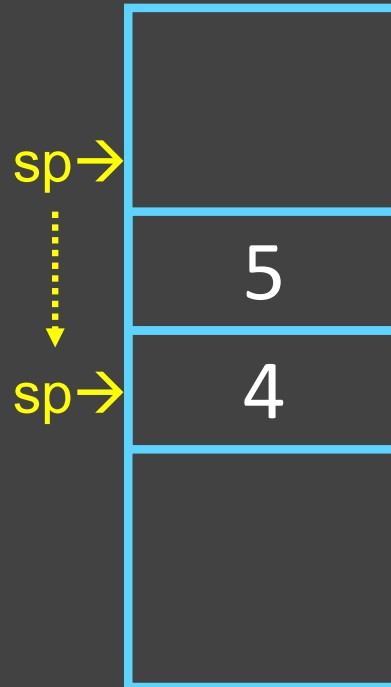
- Specifically, \$2, aka \$v0

Note: This is *not* the entire story for 1-4 arguments.  
Please see *the Full Story* slides.

# Many Arguments (5+ args)

```
main() {  
    myfn(0,1,2,3,4,5);  
    ...  
}
```

```
main:  
    li $a0, 0  
    li $a1, 1  
    li $a2, 2  
    li $a3, 3  
    addiu $sp,$sp,-8  
    li $8, 4  
    sw $8, 0($sp)  
    li $8, 5  
    sw $8, 4($sp)  
    jal myfn
```



First four arguments:  
passed in \$4-\$7

- aka \$a0-\$a3

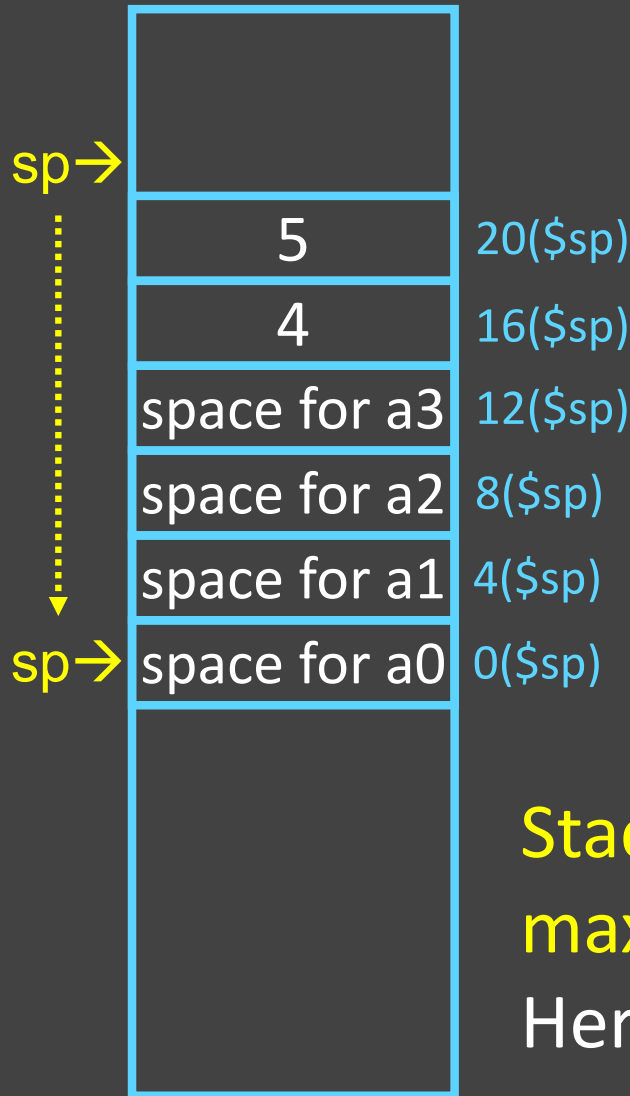
Subsequent arguments:  
"spill" onto the stack

Note: This is *not* the entire story for 5+ arguments.  
Please see *the Full Story* slides.

# Argument Passing: *the Full Story*

```
main() {  
    myfn(0,1,2,3,4,5);  
    ...  
}
```

```
main:  
    li $a0, 0  
    li $a1, 1  
    li $a2, 2  
    li $a3, 3  
    addiu $sp,$sp,-24  
    li $8, 4  
    sw $8, 16($sp)  
    li $8, 5  
    sw $8, 20($sp)  
    jal myfn
```



**Arguments 1-4:**  
passed in \$4-\$7  
*room on stack*

**Arguments 5+:**  
placed on stack

**Stack decremented by**  
 **$\max(16, \text{\#args} \times 4)$**   
Here:  $\max(16, 24) = 24$

# Pros of Argument Passing Convention

- Consistent way of passing arguments to and from subroutines
- Creates single location for all arguments
  - Caller makes room for \$a0-\$a3 on stack
  - Callee must copy values from \$a0-\$a3 to stack
    - callee may treat all args as an array in memory
  - Particularly helpful for functions w/ variable length inputs: `printf("Scores: %d %d %d\n", 1, 2, 3);`
- Aside: not a bad place to store inputs if callee needs to call a function (your input cannot stay in \$a0 if you need to call another function!)

# C & MIPS: the fine print

C allows passing whole structs

- `int dist(struct Point p1, struct Point p2);`
- Treated as collection of consecutive 32-bit arguments
  - Registers for first 4 words, stack for rest
- **Better:** `int dist(struct Point *p1, struct Point *p2);`

Where are the arguments to:

```
void sub(int a, int b, int c, int d, int e);  
void isalpha(char c);  
void treesort(struct Tree *root);
```

Where are the return values from:

```
struct Node *createNode();  
struct Node mynode();
```

Many combinations of char, short, int, void \*, struct, *etc.*

- MIPS treats char, short, int and void \* identically

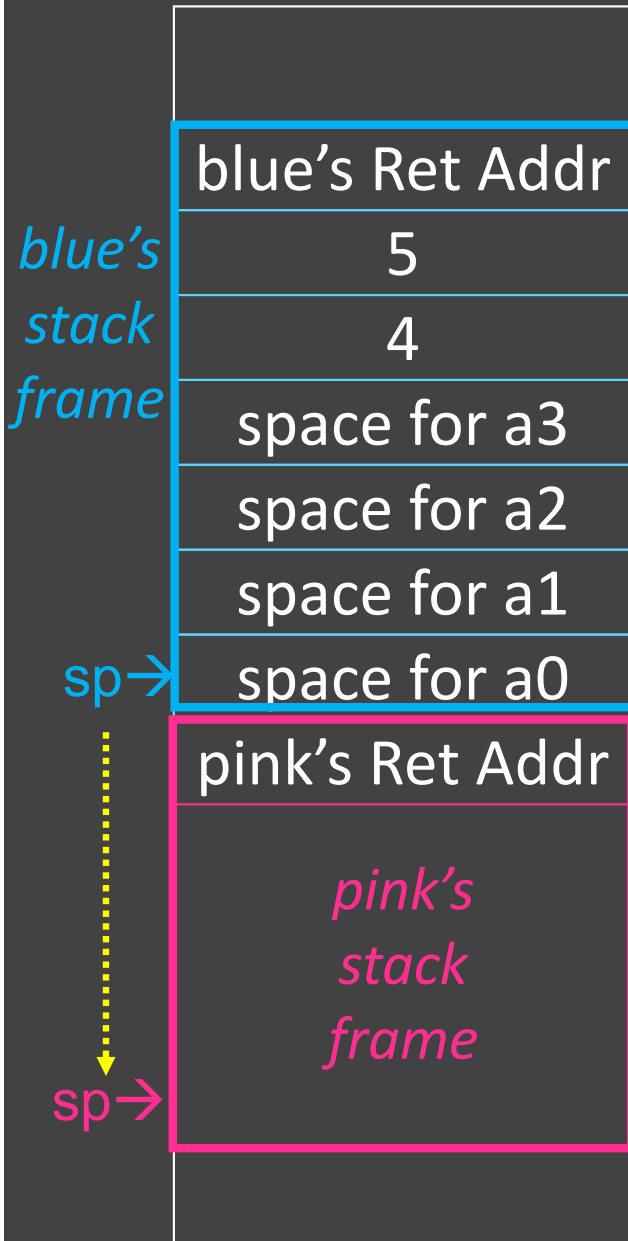
# iClicker Question

Which is a true statement about the arguments to the function

```
void sub(int a, int b, int c, int d, int e);
```

- A. Arguments a-e are all passed in registers.
- B. Arguments a-e are all stored on the stack.
- C. Only e is stored on the stack, but space is allocated for all 5 arguments.
- D. Only a-d are stored on the stack, but space is allocated for all 5 arguments.

# Frame Layout & the Frame Pointer



## Notice

- Pink's arguments are on blue's stack
- sp changes as functions call other functions, complicates accesses

→ Convenient to keep pointer to bottom of stack == **frame pointer**  
**\$30, aka \$fp**

← fp can be used to restore \$sp on exit

```
blue() {  
    pink(0,1,2,3,4,5);  
}  
pink(int a, int b, int c, int d, int e, int f) {  
    ...  
}
```

# Calling Convention for Procedure Calls

## ~~Transfer Control~~

- ~~• Caller → Routine~~
- ~~• Routine → Caller~~

## ~~Pass Arguments to and from the routine~~

- ~~• fixed length, variable length, recursively~~
- ~~• Get return value back to the caller~~

## Manage Registers

- Allow each routine to use registers
- Prevent routines from clobbering each others' data

# Register Management

Functions:

- Are compiled in isolation
- Make use of general purpose registers
- Call other functions in the middle of their execution
  - These functions also use general purpose registers!
  - No way to coordinate between caller & callee

→ Need a convention for register management

# Caller-saved

Registers that the caller cares about: \$t0... \$t9

About to call a function?

- Need value in a t-register after function returns?
  - save it to the stack before fn call
  - restore it from the stack after fn returns
- Don't need value? → do nothing

Suppose:

\$t0 holds x

\$t1 holds y

\$t2 holds z

Where do we save and restore?

## Functions

- Can freely use these registers
- Must assume that their contents are destroyed by other functions

```
void myfn(int a) {  
    int x = 10;  
    int y = max(x, a);  
    int z = some_fn(y);  
    return (z + y);  
}
```

# Callee-saved

Registers a function intends to use: \$s0... \$s9

About to use an s-register? You MUST:

- Save the current value on the stack before using
- Restore the old value from the stack before fn returns

Suppose:

\$t0 holds x

\$s1 holds y

\$s2 holds z

## Functions

- Must save these registers before using them
- May assume that their contents are preserved even across fn calls

Where do we save and restore?

```
void myfn(int a) {  
    int x = 10;  
    int y = max(x, a);  
    int z = some_fn(y);  
    return (z + y);  
}
```

# Caller-Saved Registers in Practice

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, use with no overhead

Since subroutines will do the same, must protect values needed later:

Save before fn call

Restore after fn call

Notice: Good registers to use if you don't call too many functions or if the values don't matter later on anyway.

# Callee-Saved Registers in Practice

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

Save on entry

Restore on exit

Notice: Good registers to use if you make a lot of function calls and need values that are preserved across all of them.

Also, good if caller is actually using the registers, otherwise the save and restores are wasted. But hard to know this.

# Convention Summary

- first four arg words passed in \$a0-\$a3
- remaining args passed in parent's stack frame
- return value (if any) in \$v0, \$v1
- stack frame (\$fp to \$sp) contains:
  - \$ra (clobbered on JALs)
  - local variables
  - space for 4 arguments to Callees
  - arguments 5+ to Callees
- callee save regs: preserved
- caller save regs: not preserved
- global data accessed via \$gp

\$fp →

saved ra

saved fp

saved regs  
(\$s0 ... \$s7)

locals

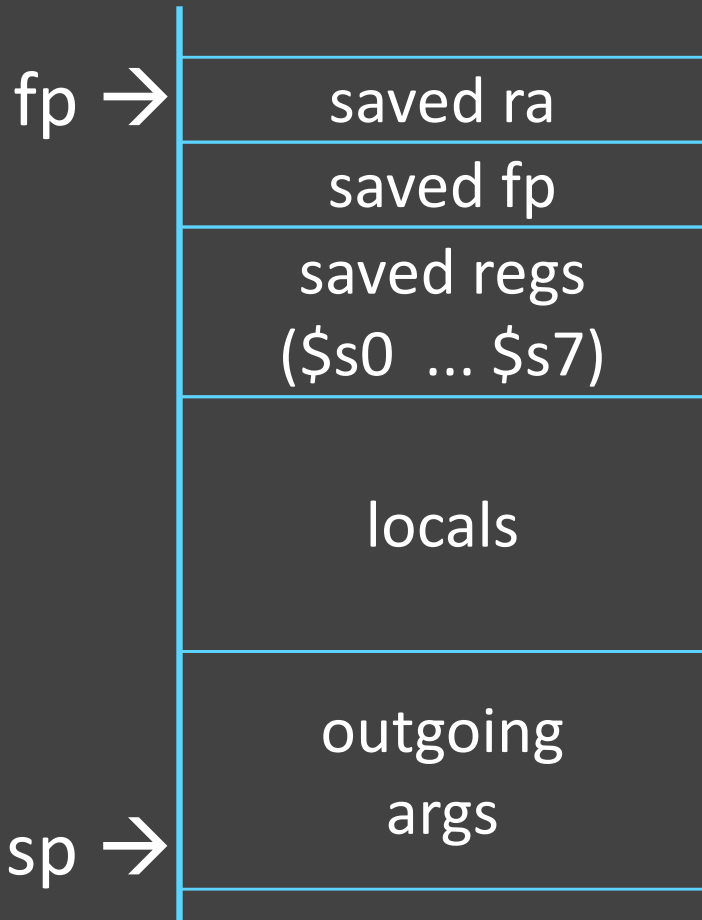
outgoing  
args

\$sp →

# MIPS Register Conventions

r0	\$zero	zero	r16	\$s0	<b>saved (callee save)</b>
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	<b>temps (caller save)</b>	r24	\$t8	<b>more temps (caller save)</b>
r9	\$t1		r25	\$t9	
r10	\$t2		r26	\$k0	reserved for kernel
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

# Frame Layout on Stack



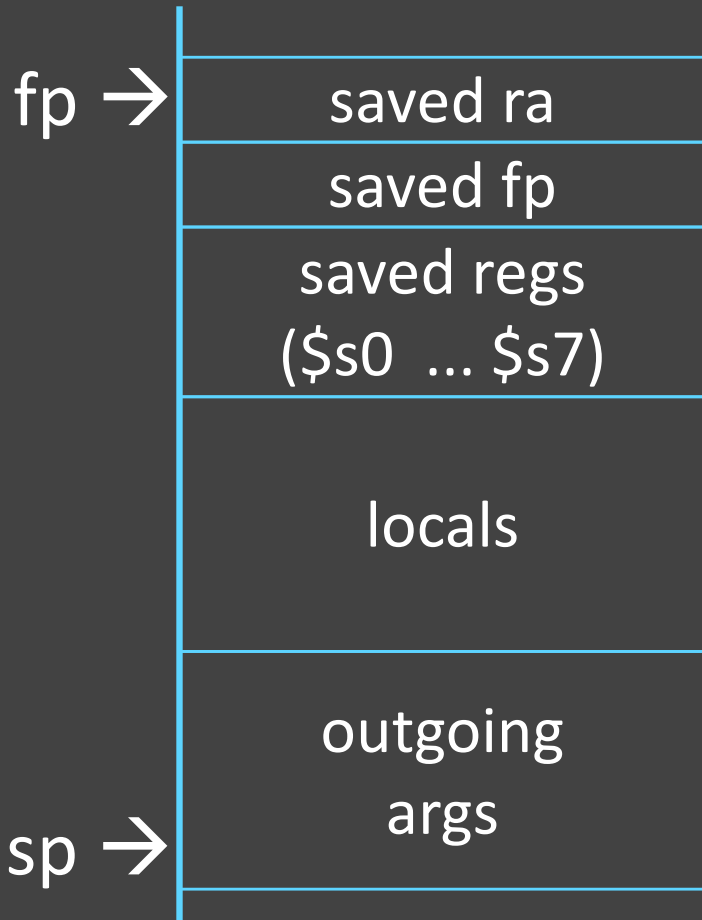
Assume a function uses two callee-save registers.

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
    
```

# Buffer Overflow



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

```
pink(int a, int b, int c, int d, int e, int f) {  
    int x;  
    orange(10,11,12,13,14);  
}
```

```
orange(int a, int b, int c, int, d, int e) {  
    char buf[100];  
    gets(buf);    // no bounds check!  
}
```

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

# Optimizing Leaf Functions

*Leaf function* does not invoke any other functions

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

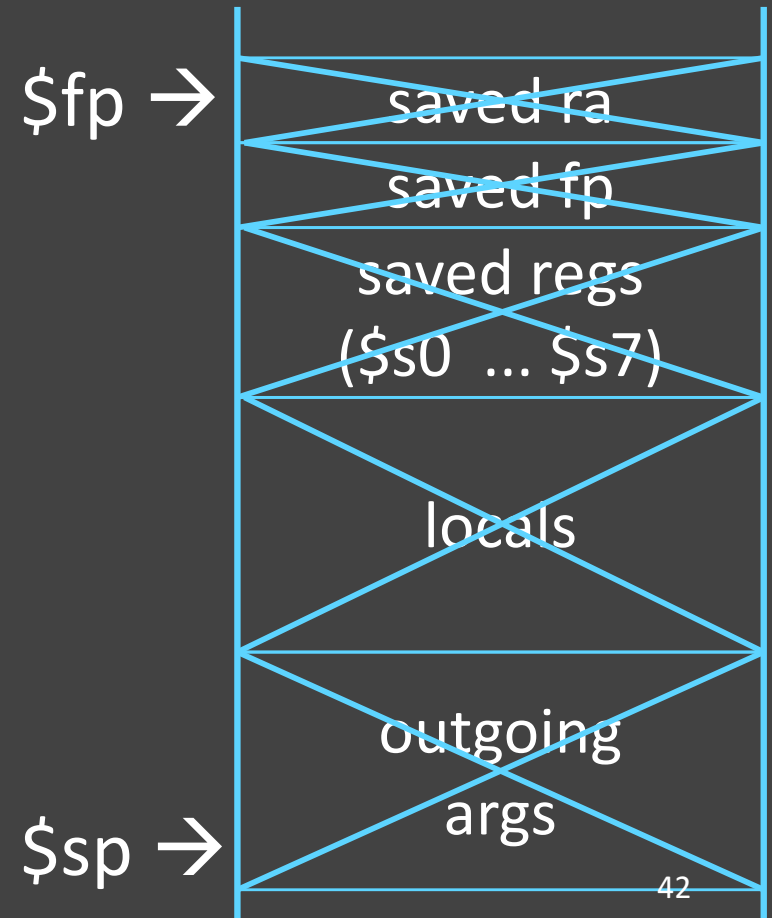
Optimizations?

No saved regs (or locals)

No outgoing args

Don't push \$ra

No frame at all? *Possibly...*



# Activity #1: Body



```
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;  
}
```

## Correct Order:

1. Body First
2. Determine stack frame size
3. Complete Prologue/Epilogue

# Activity #1: Body

```
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;  
}
```

*We'll assume the yellow in order to force your hand on the rest.*

\$s0 for \$a0 / a

\$s1 for \$a1 / b

\$t0 for tmp

Can we get rid of the NOP?

We want to do the lw...

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)
```

# Activity #1: Body

```
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;  
}
```

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

```
ADD $v0, $v0, $s0    # u + a  
ADD $v0, $v0, $s1    # + b
```

Epilogue

## Activity #2: Frame Size



```
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) 36
- c) 44
- d) 48
- e) 52

Clicker  
Question

Minimum stack size for a standard function?

saved ra
saved fp
saved regs (\$s0 and \$s1)
locals (\$t0)
outgoing args space for a0 - a3 and 5 <sup>th</sup> and 6 <sup>th</sup> arg

# Activity #2: Frame Size

```
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?

44

Minimum stack size for a standard function?

$\$ra + \$fp + 4 \text{ args} = 6 \times 4 \text{ bytes} = 24 \text{ bytes}$

saved ra
saved fp
saved regs (\$s0 and \$s1)
locals (\$t0)
outgoing args space for a0 - a3 and 5 <sup>th</sup> and 6 <sup>th</sup> arg

fp → 40	saved ra
36	saved fp
32	saved reg \$s1
28	saved reg \$s0
24	local \$t0
20	outgoing 6 <sup>th</sup> arg
16	outgoing 5 <sup>th</sup> arg
12	space for \$a3
8	space for \$a2
4	space for \$a1
sp → 0	space for \$a0

# Activity #3: Prologue & Epilogue



```
# 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 #3: Prologue & Epilogue

fp →	40	saved ra
	36	saved fp
	32	saved reg \$s1
	28	saved reg \$s0
	24	local \$t0
	20	outgoing 6 <sup>th</sup> arg
sp →	16	outgoing 5 <sup>th</sup> arg
	12	space for \$a3
	8	space for \$a2
	4	space for \$a1
	0	space for \$a0

```

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
                        ...
                        ...
                        # restore ...
                        # restore ...
                        # restore old $fp
                        # restore $ra
                        # dealloc frame
JR $ra
NOP
    
```

Body

(previous slide, Activity #1)