Calling Conventions

Anne Bracy

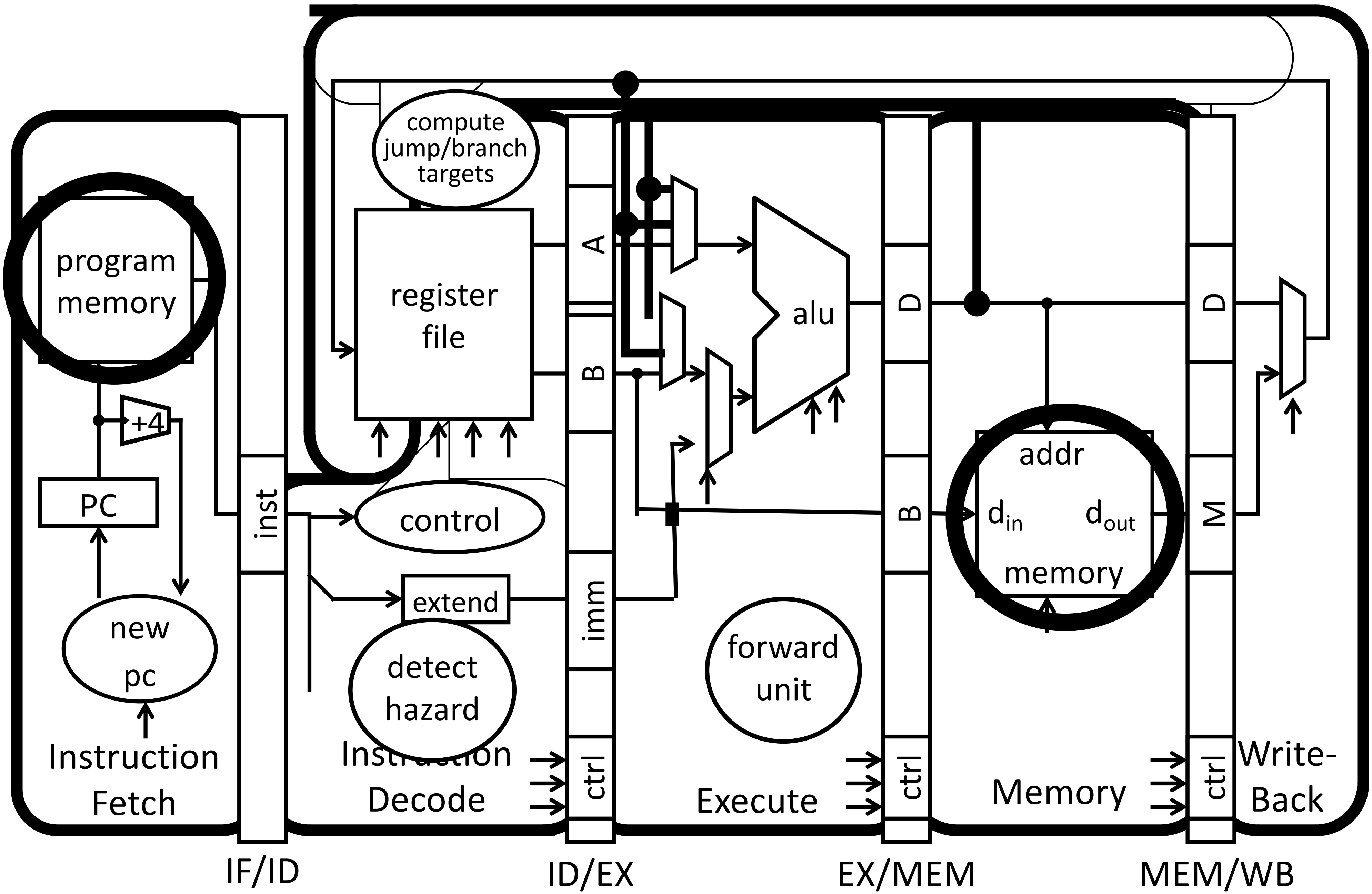
CS 3410

Computer Science

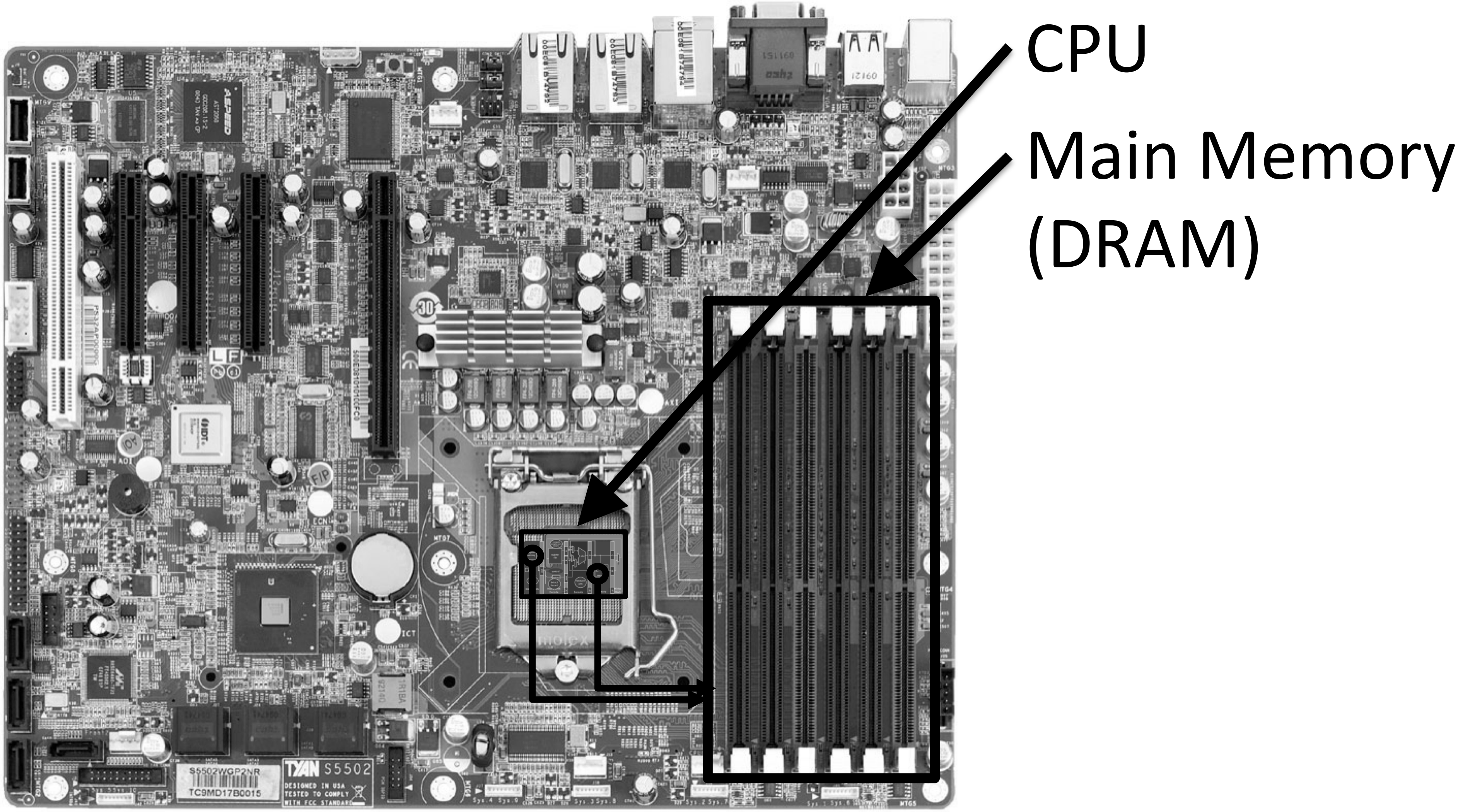
Cornell University

[K. Bala, A. Bracy, E. Sirer, and H. Weatherspoon]

An executing program on chip

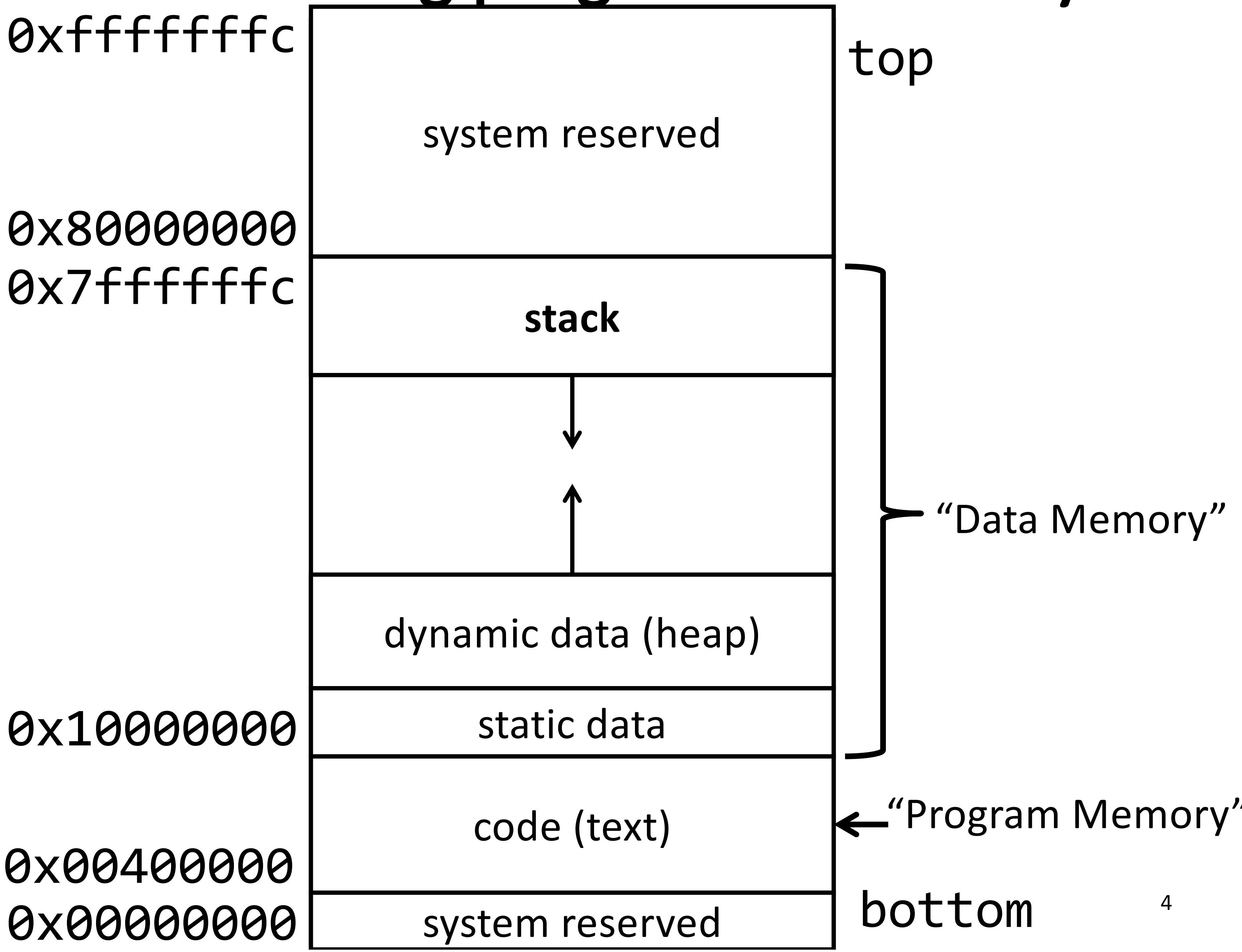


Where is Memory?



SandyBridge Motherboard, 2011
http://news.softpedia.com

An executing program in memory



The Stack

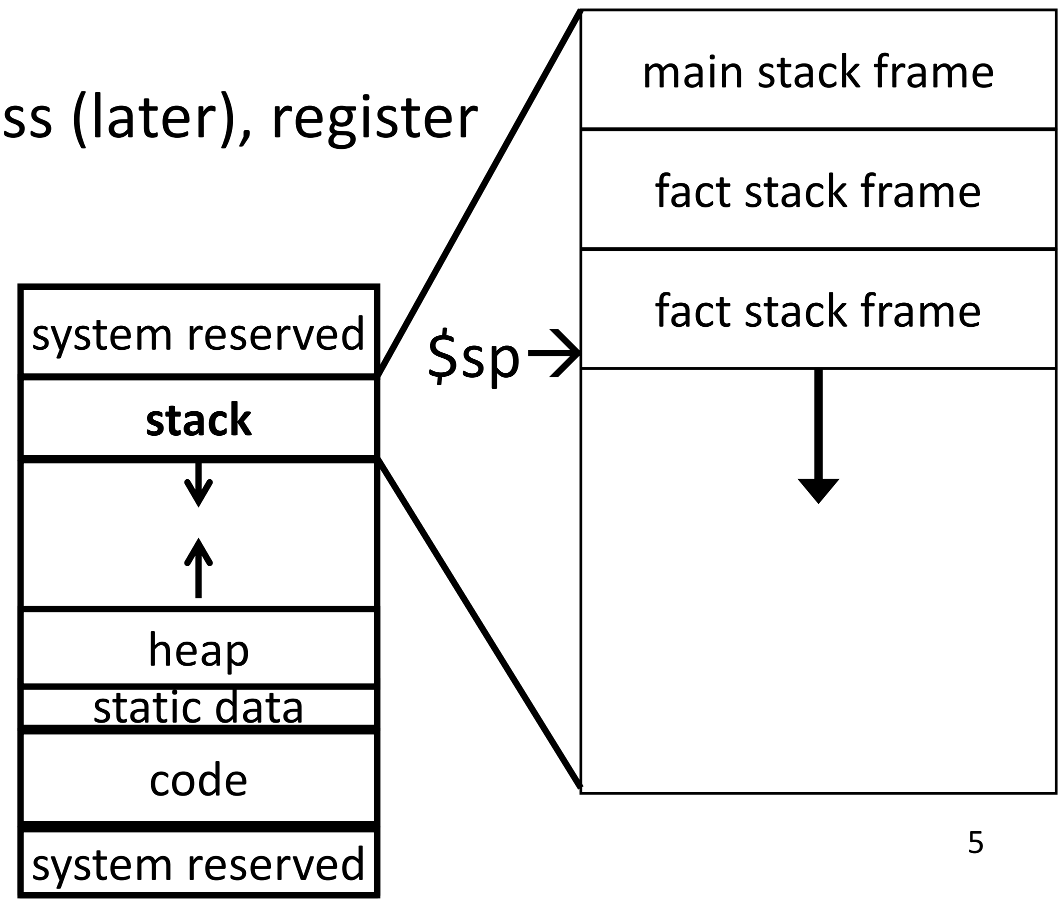
Stack contains stack frames

- 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(…) {
     …
     fact(x);
}
int fact(int n) {
     …
     fact();
}
```

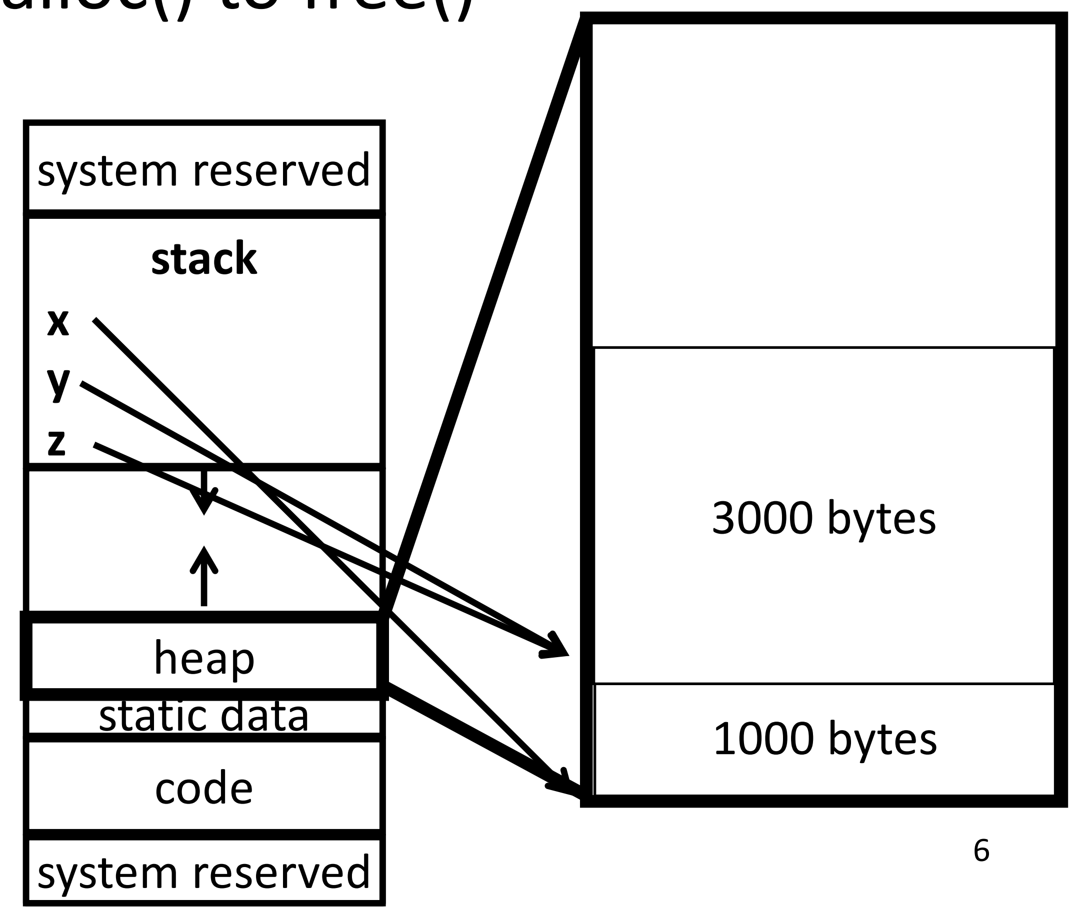


The Heap

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
- Request is valid from malloc() to free()

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



Data Segment

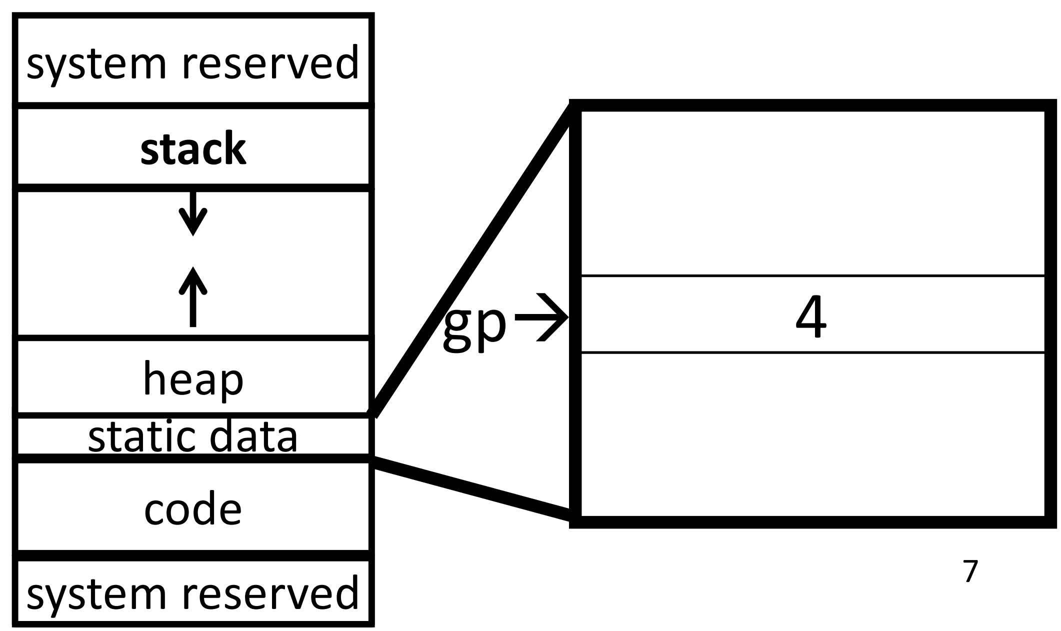
Data segment contains global variables

- 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);
}
```

Where is `main` ?

(A) Stack
(B) Heap
(C) Global Data
(D) Text

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);
}
```

How does a function call work?

```
int main (int argc, char* argv[ ]) {
     int n = atoi(argv[1]);
     int result = fact(n);
     printf(“factorial of %d is %d\n”, n, result);
     n--;
     result = fact(n);
     printf(“factorial of %d is %d\n”, n, result);
     return 0;
}

int fact(int n) {
      if (n == 0)
         return 1;
      else
         return(n * fact(n-1));
}
```

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

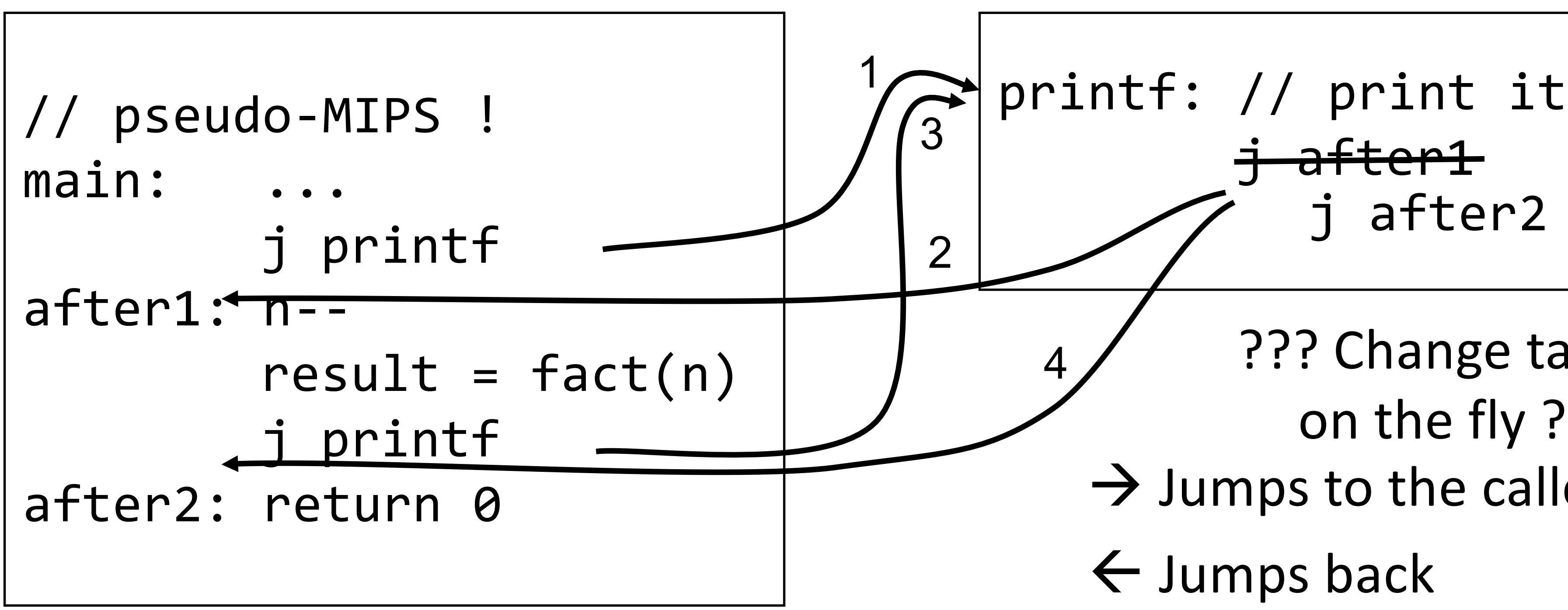
What is a Convention?

Warning: There is no one true MIPS calling convention.

lecture != book != gcc != spim != web

Jumps are not enough

```
int main (int argc, char* argv[ ]) {
      int n = atoi(argv[1]);
      int result = fact(n);
      printf("fact of %d is %d\n", n, result);
      n--;
      result = fact(n);
      printf("fact of %d is %d\n", n, result);
      return 0;
}
```



??? Change target on the fly ???

→ Jumps to the callee

← Jumps back

Jump-and-Link / Jump Register

```
int main (int argc, char* argv[ ]) {
      int n = atoi(argv[1]);
      int result = fact(n);
      printf("fact of %d is %d\n", n, result);
      n--;
      result = fact(n);
      printf("fact of %d is %d\n", n, result);
      return 0;
}
```

r31 `after1`

First Call

```
// pseudo-MIPS !
main:    ...
         jal printf
after1:  n--
         result = fact(n)
         jal printf
after2:  return 0
```

```
printf: // print it
        jr $31
```

1. Jumps to callee, save PC+4
2. Jumps back to saved PC+4

Jump-and-Link / Jump Register

```
int main (int argc, char* argv[ ]) {
      int n = atoi(argv[1]);
      int result = fact(n);
      printf(“fact of %d is %d\n”, n, result);
      n--;
      result = fact(n);
      printf(“fact of %d is %d\n”, n, result);
      return 0;
}
```

r31 `after2`

Second Call

```
// pseudo-MIPS !
main:    ...
         jal printf
after1:  n--
         result = fact(n)
         jal printf
after2:  return 0
```

```
printf: // print it
        jr $31
```

3. Jumps to the callee, save PC+4
4. Jumps back to saved PC+4

JAL/JR with Recursion?

```
int main () {
      ...
      int result = fact(n);
      ...
}
```

```
int fact(int n) {
      if (n == 0)
         return 1;
      else
         return(n * fact(n-1));
}
```

r31 `after1`

First Call

```
// pseudo-MIPS !
main:   ...
        jal printf
after1: n--
        ...
```

1

```
fact: bne r0,r1, rec
      jr $31
rec:  subi r1,r1,1
      jal fact
loc2: mult r1,...
```

JAL/JR with Recursion?

```
int main () {
      ...
      int result = fact(n);
      ...
}
```

```
int fact(int n) {
      if (n == 0)
         return 1;
      else
         return(n * fact(n-1));
}
```

Second Call r31 `loc2`

```
// pseudo-MIPS !
main:   ...
        jal printf
after1: n--
        ...
```

1

2

```
fact: bne r0,r1, rec
      jr $31
rec:  subi r1,r1,1
      jal fact
loc2: mult r1,...
```

Overwrites contents of $31!

How do we get back to main?

Solution: Save Return Address in Stack Frame

Context: just called fact from main, `r31← after1`

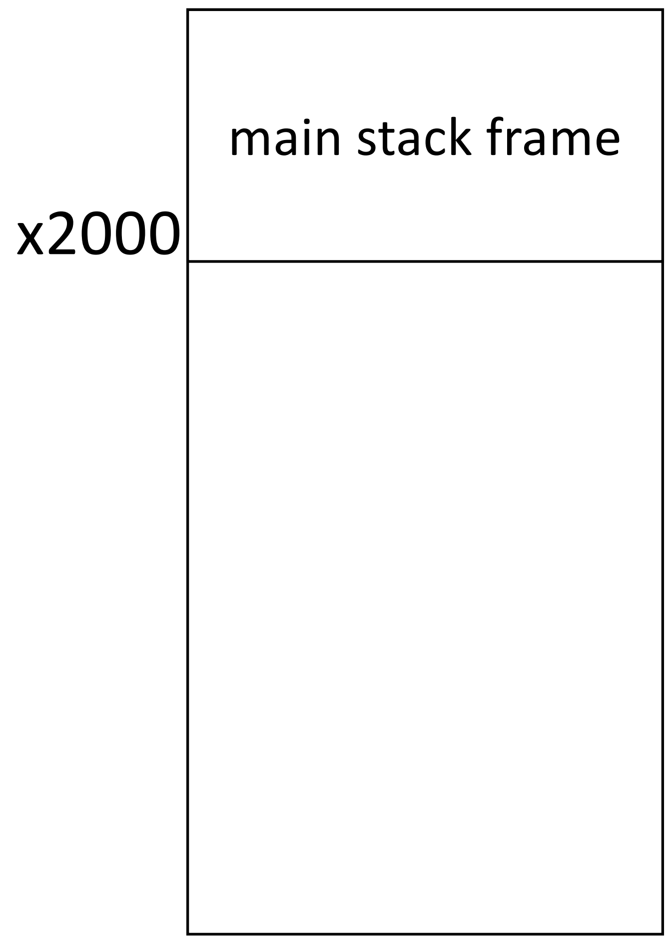
stack ptr, $sp: r29 = x2000

return address: r31 = after1

```
main:   ...
        jal printf
after1: n--
        ...
```

1

```
fact:
        bne r0,r1, rec
        jr $31
rec:    subi r1,r1,1
        jal fact
loc2:   mult r1,...
```

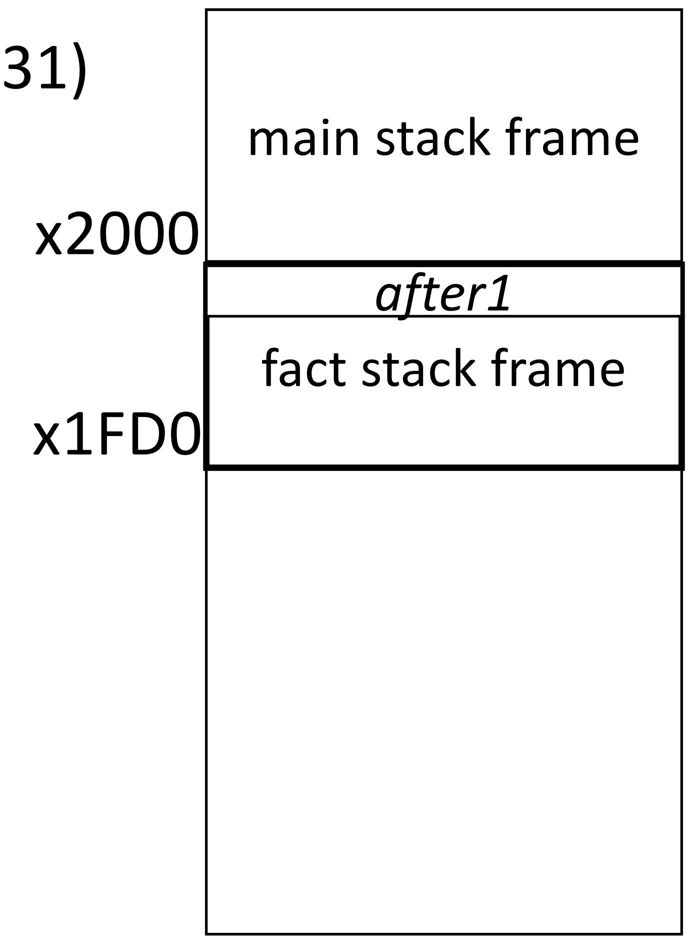


Solution: Save Return Address in Stack Frame

Context: just called fact from main, `r31← after1`

PUSH: ADDIU $sp, $sp, -20 // move sp down*

SW $31, 16($sp) // store retn PC (r31)



stack ptr, $sp: r29 `x1FD0`

return address: r31 `after1`

```
main:    ...
         jal printf
after1:  n--
         ...
```

1

```
fact:  PUSH
       bne r0,r1, rec
       jr $31
rec:   subi r1,r1,1
       jal fact
loc2:  mult r1,...
```

**say each frame is x20 bytes*

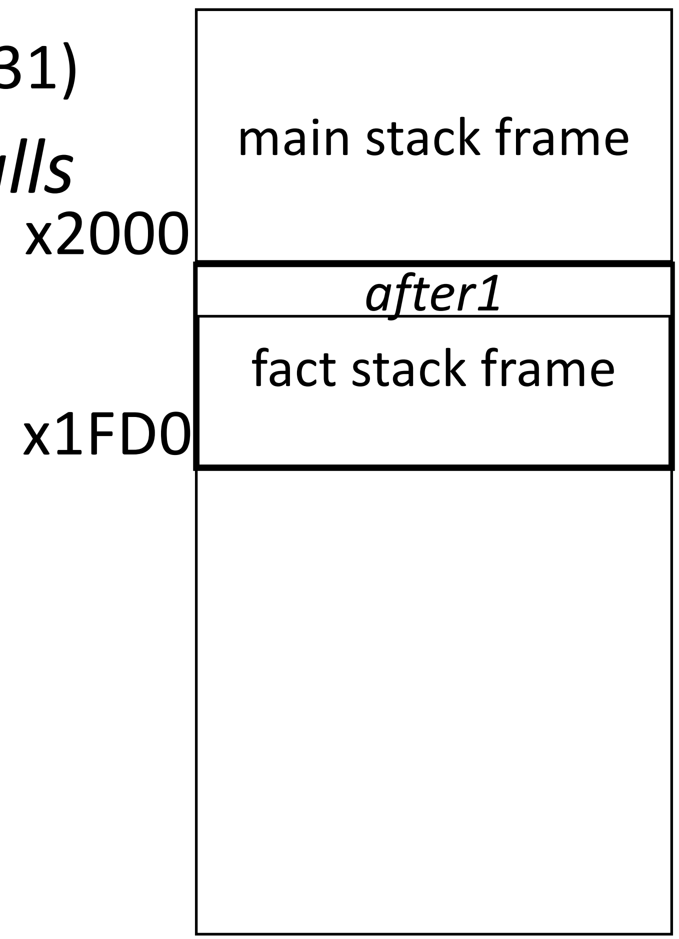
Solution: Save Return Address in Stack Frame

Context: just called fact from main, `r31← after1`

PUSH: ADDIU $sp, $sp, -20 // move sp down
SW $31, 16($sp) // store retn PC (r31)

Fast-forward past a bunch of recursive calls

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



stack ptr, $sp: r29 = x2000

return address: r31 = after1

```
main:    ...
         jal printf
after1:  n--
         ...
```

```
fact:  PUSH
       bne r0,r1, rec
       jr $31 POP
rec:   subi r1,r1,1
       jal fact
loc2:  mult r1,...
```

(In the code above, "jr $31" is struck through and replaced by "POP".)

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

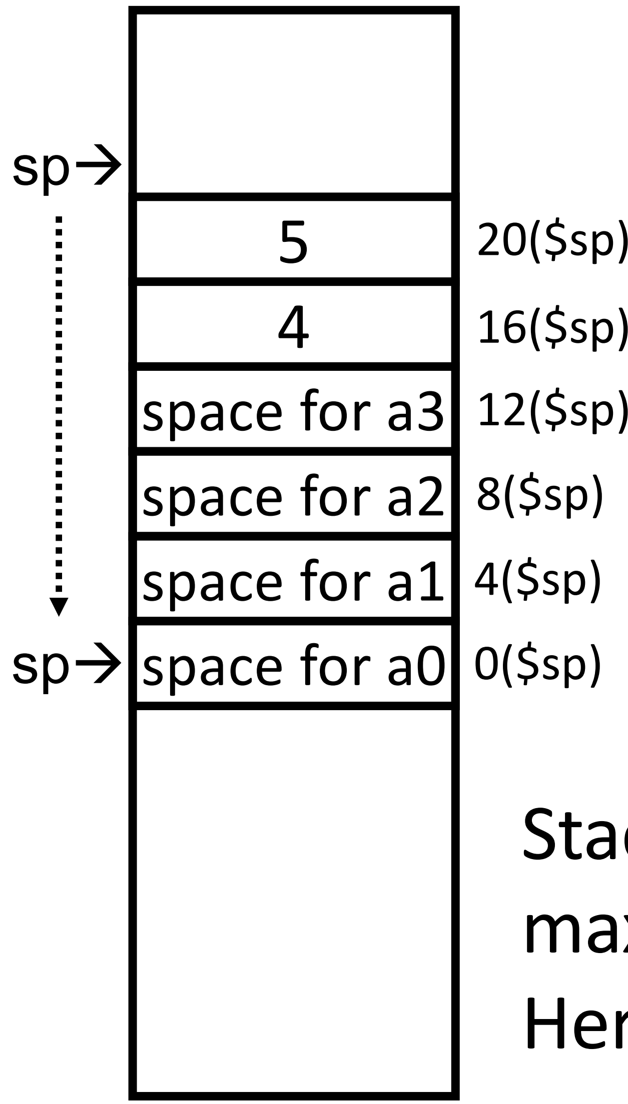
- Specifically, \$2, aka \$v0

Note: This is *not* the entire story for 1-4 arguments. See next slide…

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, #args x 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!)

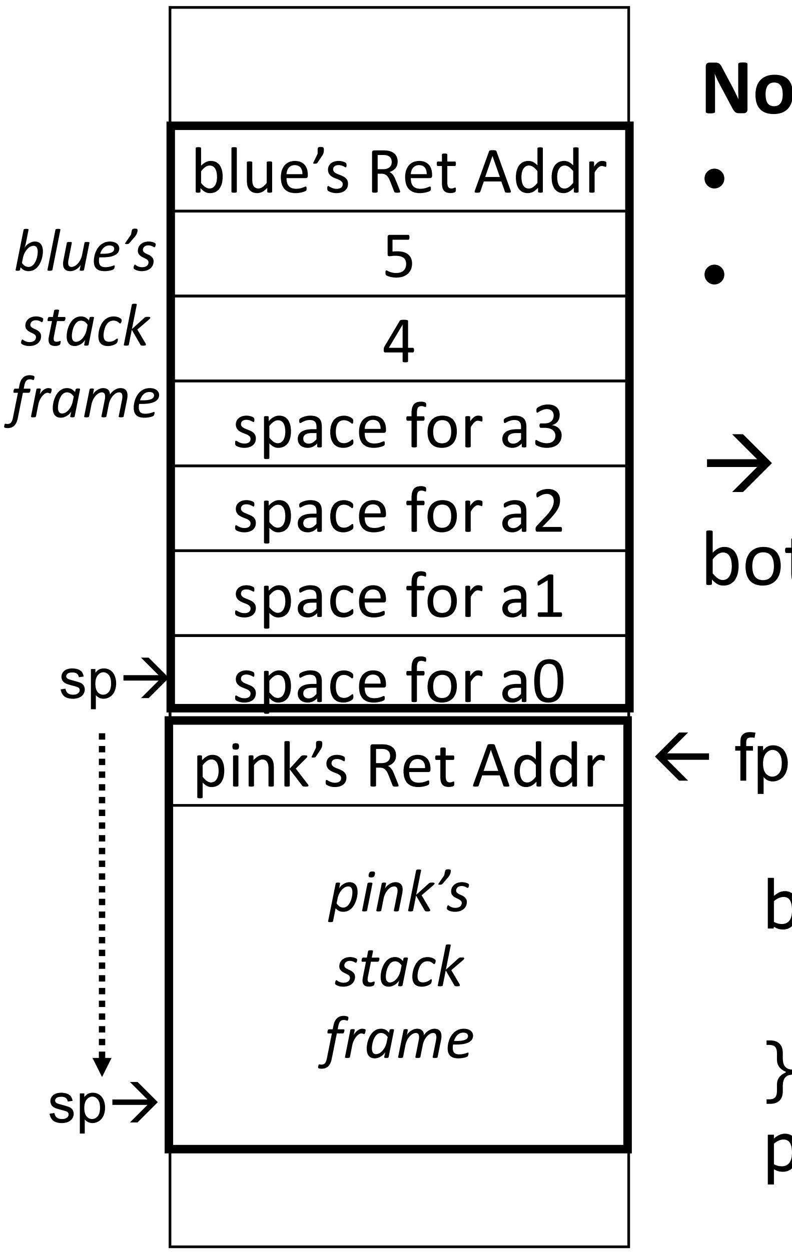
iClicker Question

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

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

Before a function call

Does caller need value in a t-register after fn returns?

Yes: → save it to the stack before fn call

→ restore it from the stack after fn returns

No: do nothing, it might get clobbered, fn doesn't care

All Functions

- Can freely use these registers
- Assume that their contents are clobbered by any called functions

A good place to put **temporary** values, hence t-registers

Caller-saved, Clicker Question 1

Registers that the caller cares about: $t0… $t9

Before a function call

Does caller need value in a t-register after fn returns?

Yes: → save it to the stack before fn call

→ restore it from the stack after fn returns

No: do nothing, it might get clobbered, fn doesn't care

<u>Suppose:</u>
$t0 holds x
$t1 holds y
$t2 holds z

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

What gets saved before calling max?

(a) t0
(b) t1
(c) t0 and t1
(d) t0,t1, and t2
(e) no t-registers

Caller-saved, Clicker Question 2

Registers that the caller cares about: $t0… $t9

Before a function call

Does caller need value in a t-register after fn returns?

Yes: → save it to the stack before fn call

→ restore it from the stack after fn returns

No: do nothing, it might get clobbered, fn doesn't care

<u>Suppose:</u>
$t0 holds x
$t1 holds y
$t2 holds z

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

What gets saved before calling some_fn?

(a) t1
(b) t2
(c) t1 and t2
(d) t0,t1, and t2
(e) no t-registers

Callee-saved

Registers a callee must preserve: $s0… $s9

About to use an s-register? A function MUST:

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

All Functions

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

Must **save** the register before using it, hence s-registers

Callee-saved, Clicker Question

Registers a callee must preserve: \$s0… \$s9

About to use an s-register? A function 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

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

What gets saved before the body of myfn?

(a) s1
(b) s2
(c) s0-s9
(d) t0,s1,s2
(e) t0,s0-s9

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]
```

t 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.

Clicker Question

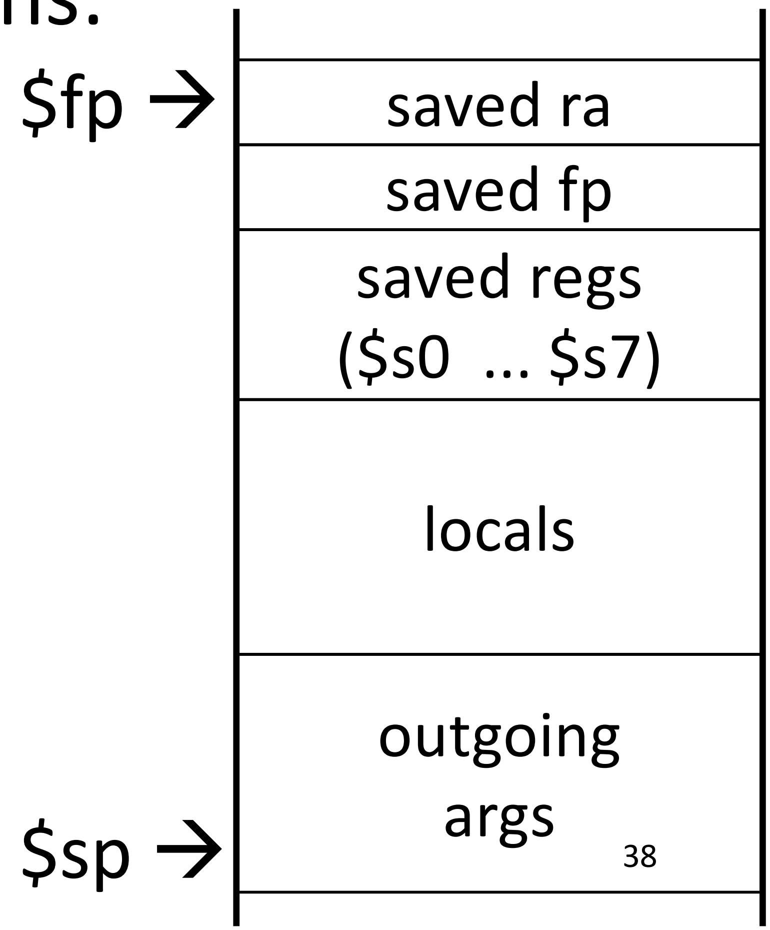
```
int foo() {
  int a = 0;
  int b = 12;
  int c = 1;
  int e = b + bar(c);
  c = b + a;
  int d = c + baz(b);
  a = d-e;
  return a;
}
```

If a compiler wanted minimize the work required to compile just this function, would it put **a** in a:

(A) Caller-saved register (t)
(B) Callee-saved register (s)
(C) Depends on where we put the other variables in this fn
(D) Both are equally good
(E) Initially a t register, then move it to an s register after baz

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

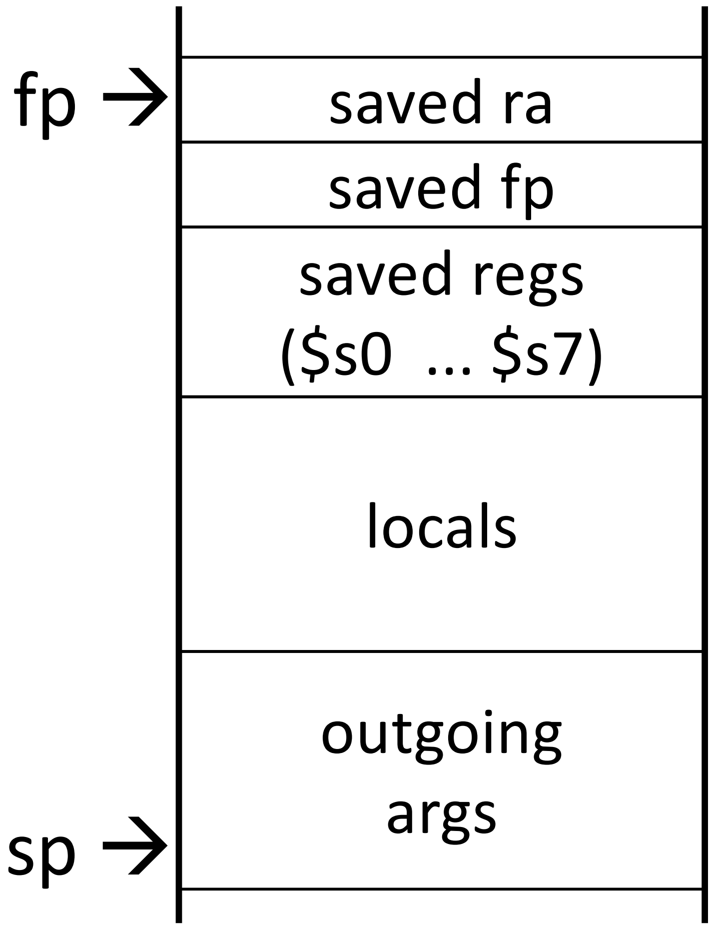


MIPS Register Conventions

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

Frame Layout on Stack



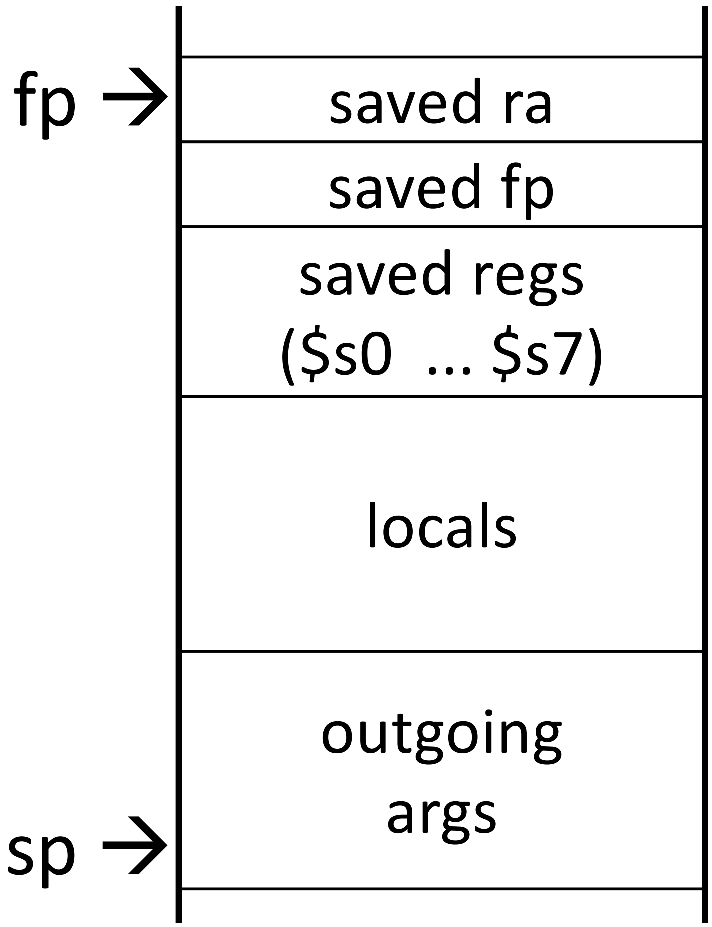
Assume a function uses two callee-save registers.

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

What should be stored in the stack frame?

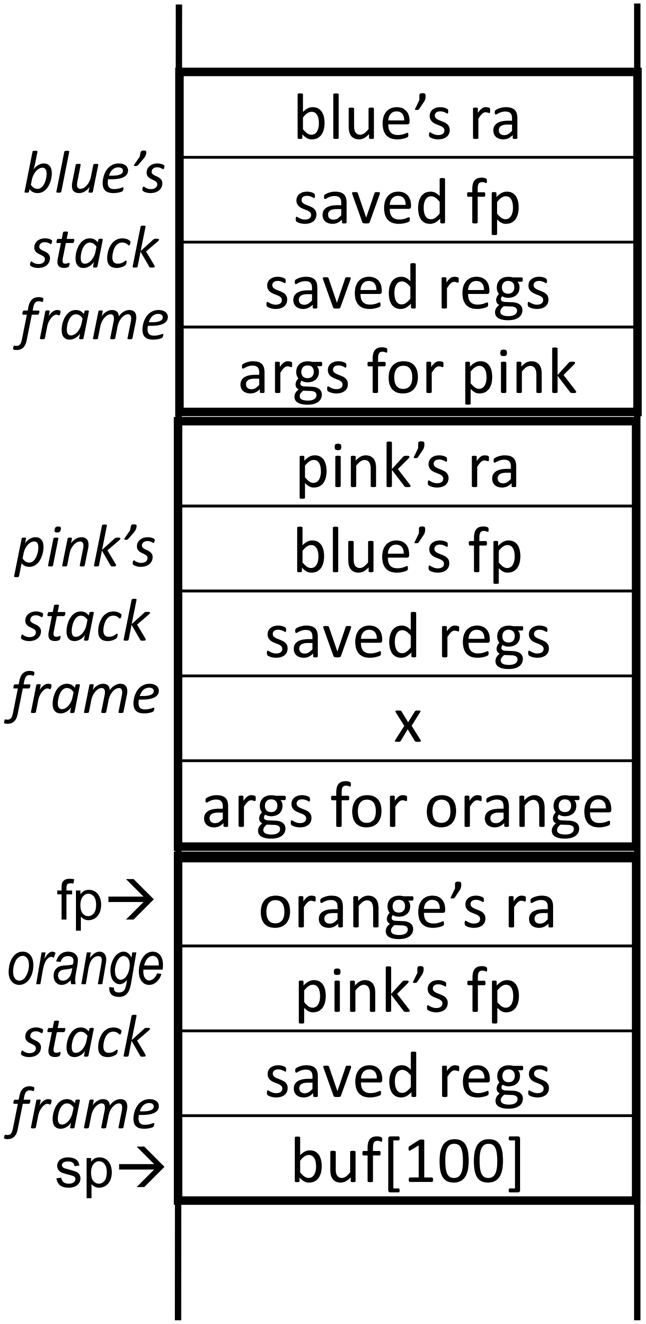
Where should everything be stored?

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?

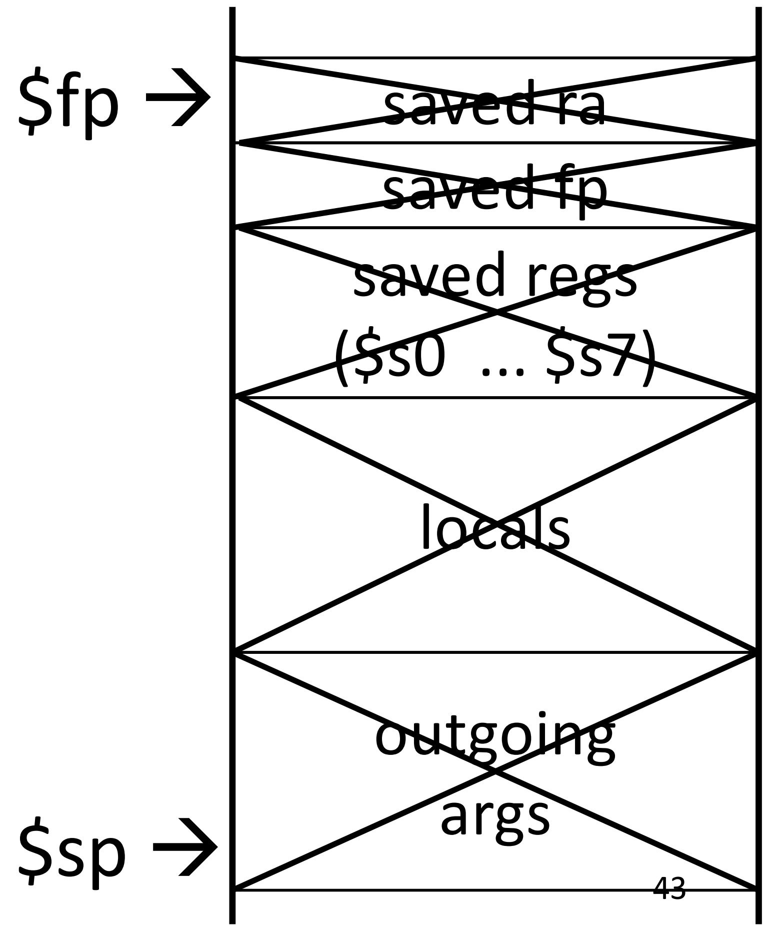
Optimizing Leaf Functions

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;
}
```

<u>Correct Order:</u>

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?

Activity #2: Frame Size

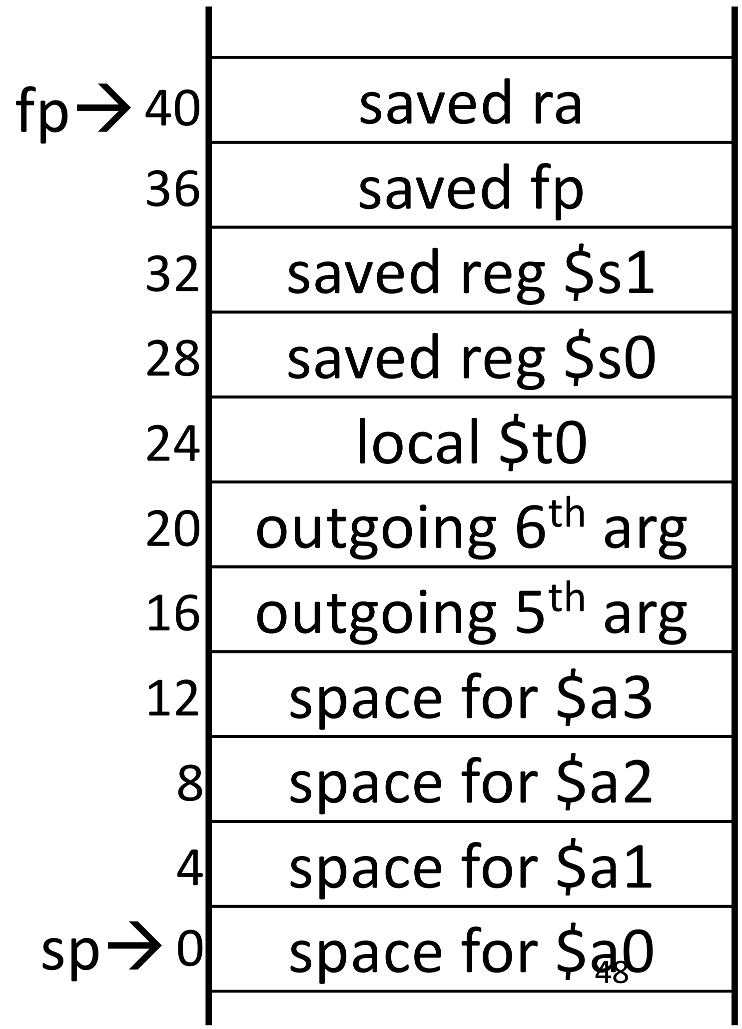
```
int test(int a, int b) {
    int tmp = (a&b)+(a|b);
    int s = sum(tmp,1,2,3,4,5);
    int u = sum(s,tmp,b,a,b,a);
    return u + a + b;
}
```

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

44

Minimum stack size for a standard function?

\$ra + \$fp + 4 args =
6 x 4 bytes = 24 bytes



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

	Offset	Contents
fp→	40	saved ra
	36	saved fp
	32	saved reg $s1
	28	saved reg $s0
	24	local $t0
	20	outgoing 6th arg
	16	outgoing 5th arg
	12	space for $a3
	8	space for $a2
	4	space for $a1
sp→	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
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
```