Calling Conventions

Hakim Weatherspoon

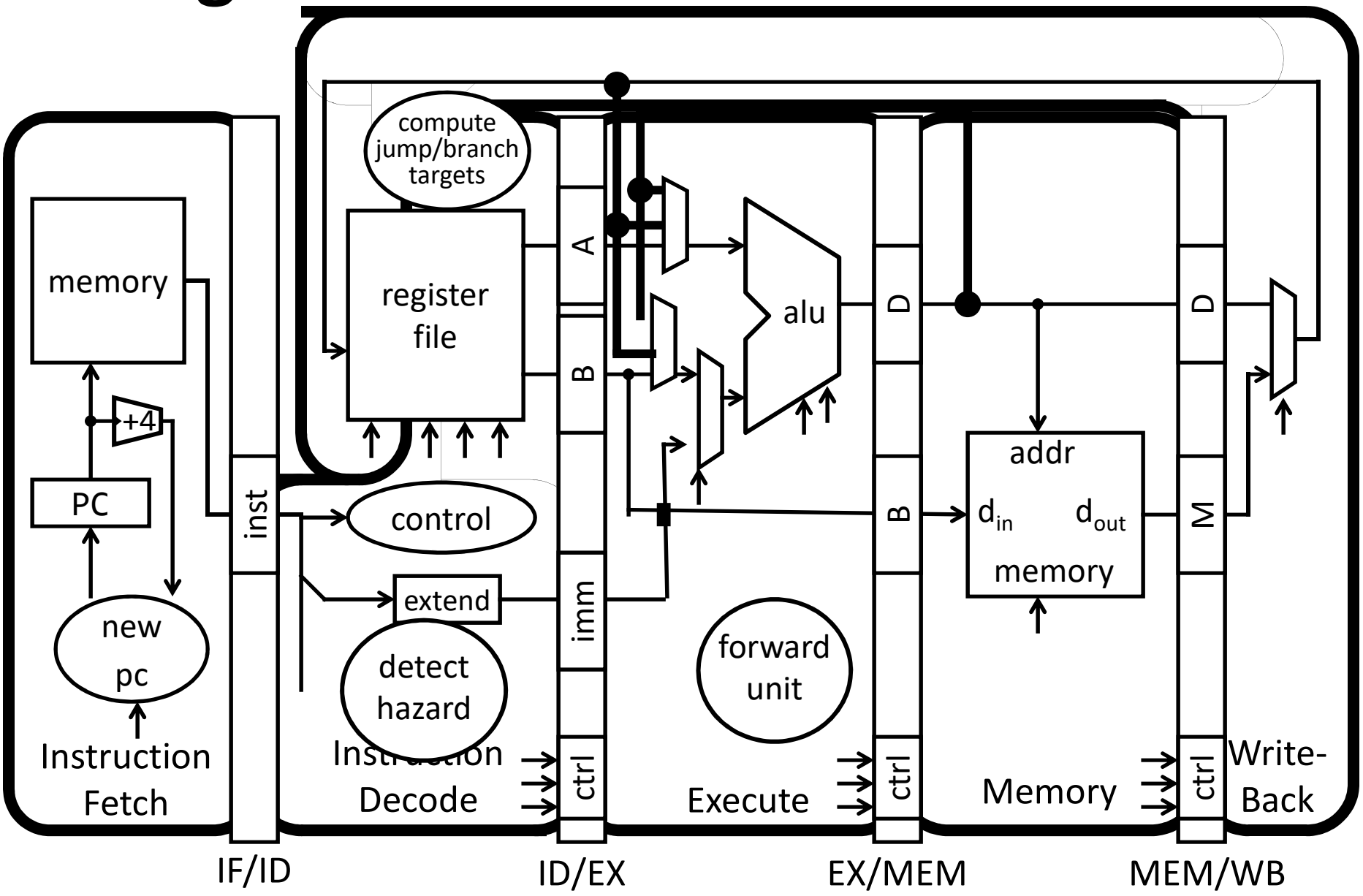
CS 3410

Computer Science

Cornell University

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

Big Picture: Where are we now?



Big Picture: Where are we going?

C

```
int x = 10;
x = 2 * x + 15;
```

compiler

MIPS assembly

```
addiu r5, r0, 10
muli  r5, r5, 2
addiu r5, r5, 15
```

r0 = 0

r5 = r0 + 10

r5 = r5<<1 #r5 = r5 * 2

r5 = r15 + 15

assembler

op = addiu r0 r5 10

machine code

```
001000 00000 00101 0000000000001010
000000 00000 00101 00101 00001 000000
001000 00101 00101 0000000000001111
```

op = addiu r5 r5 15

CPU

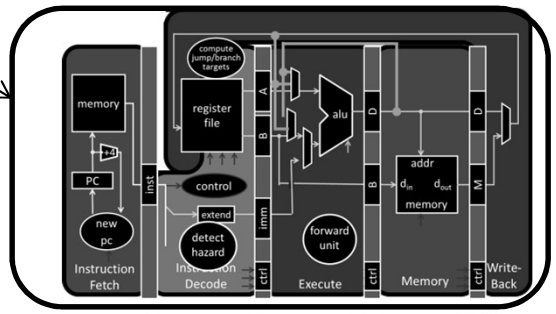
op = r-type r5 r5 shamt=1 func=sll

Circuits

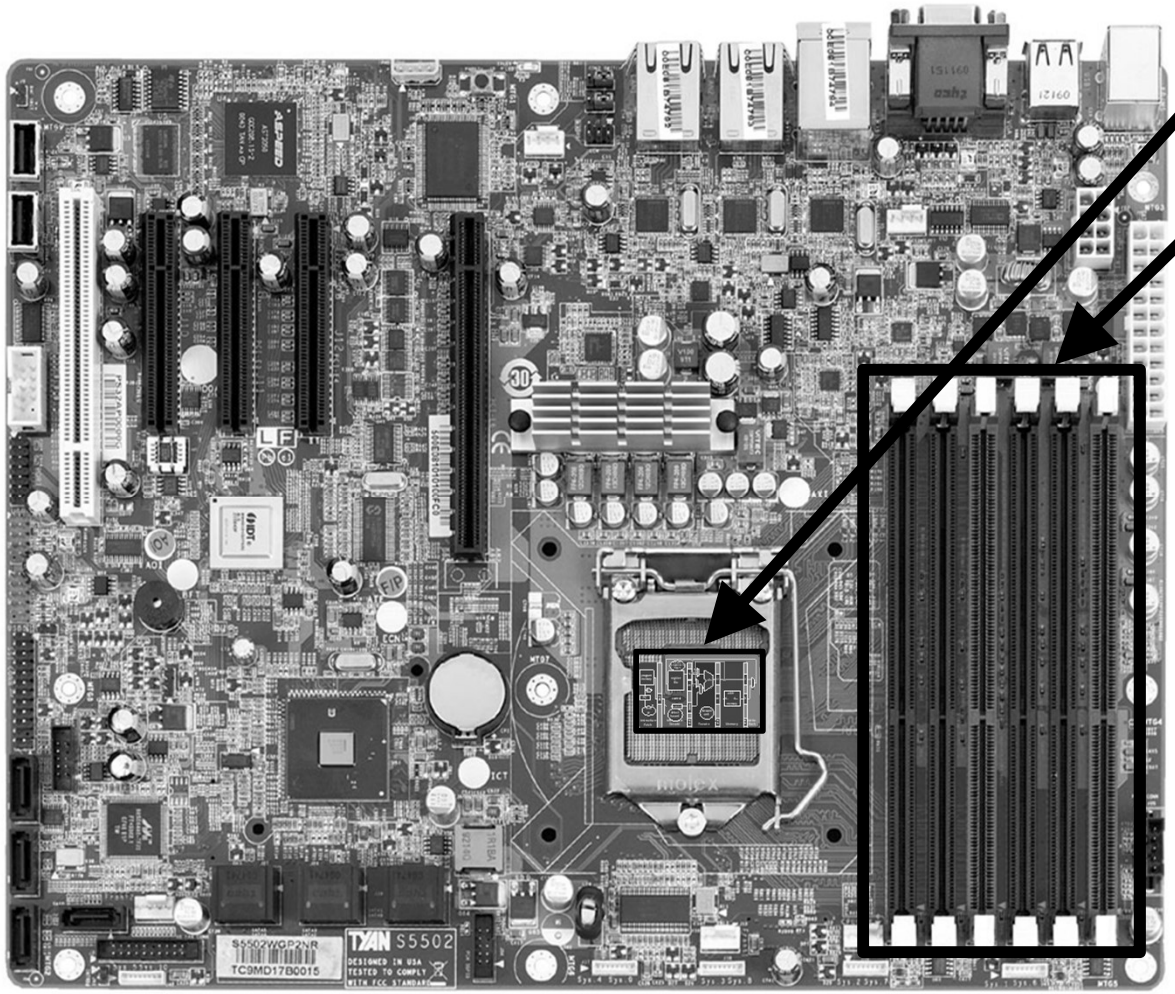
Gates

Transistors

Silicon



Big Picture: Where are we going?



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

Goals for this week

Calling Convention for Procedure Calls

Enable code to be reused by allowing code snippets to be invoked

Will need a way to

- call the 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

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

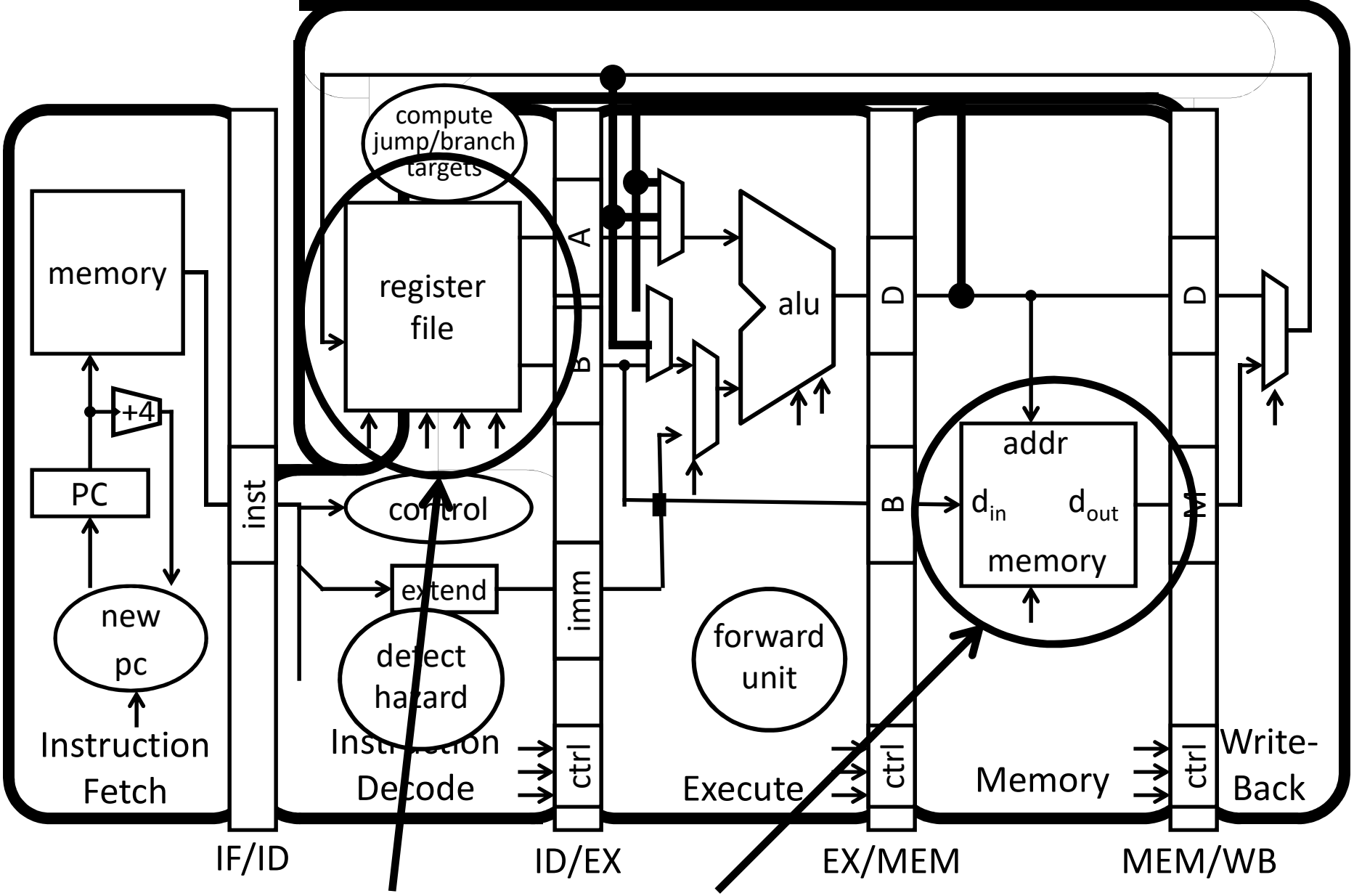
- Allow each routine to use registers
- Prevent routines from clobbering each others' data

What is a Convention?

Warning: There is no one true MIPS calling convention.

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

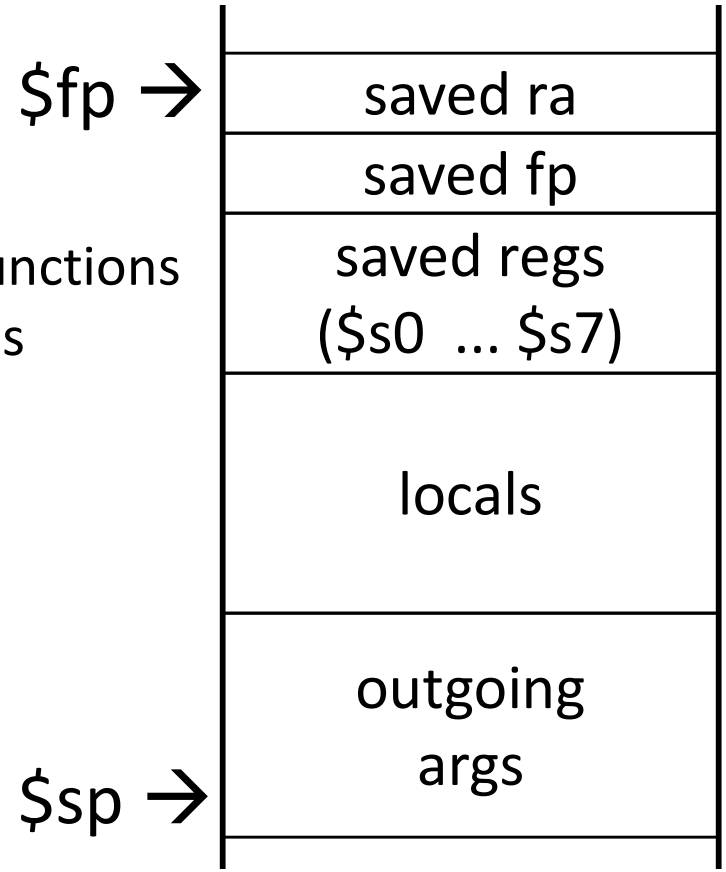
Cheat Sheet and Mental Model for Today



How do we share registers and use memory when making procedure calls?

Cheat Sheet and Mental Model for Today

- 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 local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions
  - contains space for first 4 arguments to sub-functions
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp



MIPS Register

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
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

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?

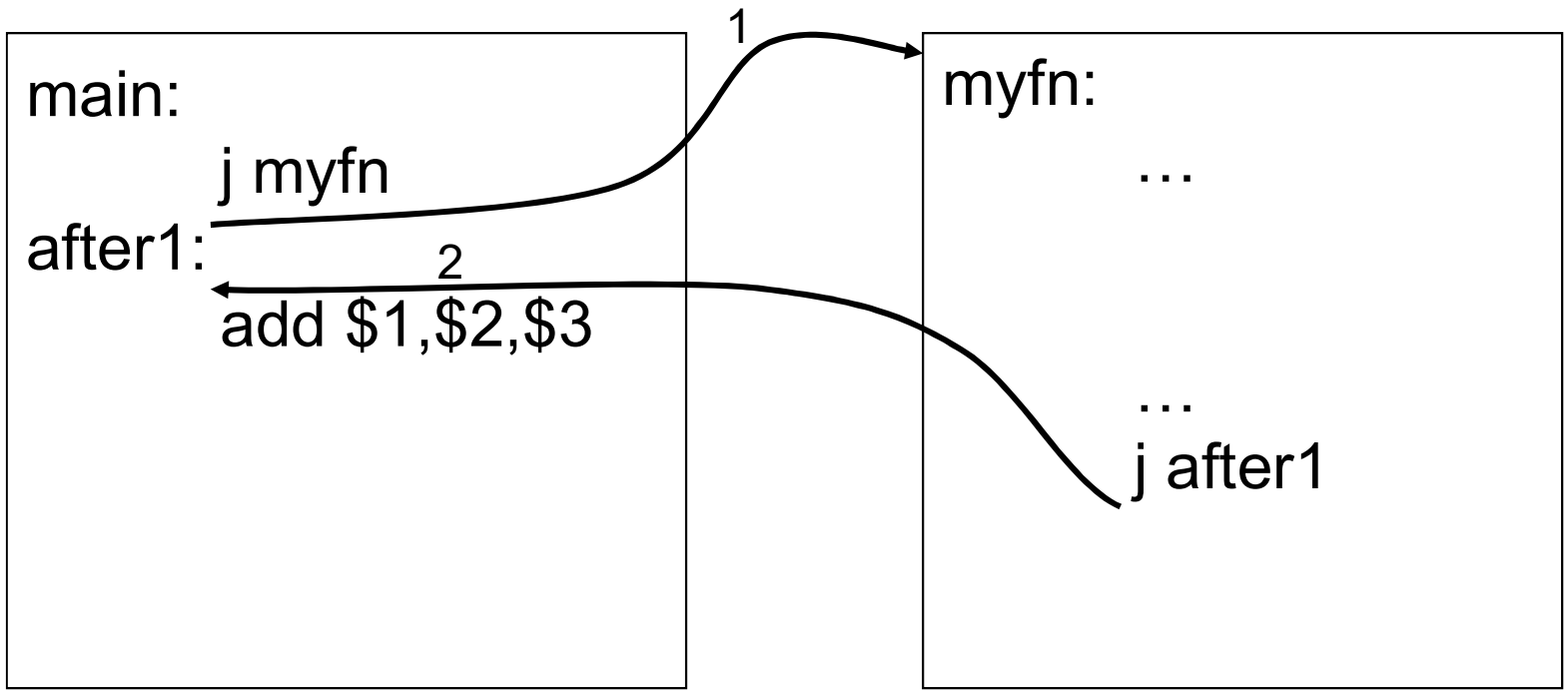
Warning: There is no one true MIPS calling convention.

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

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;
}
```

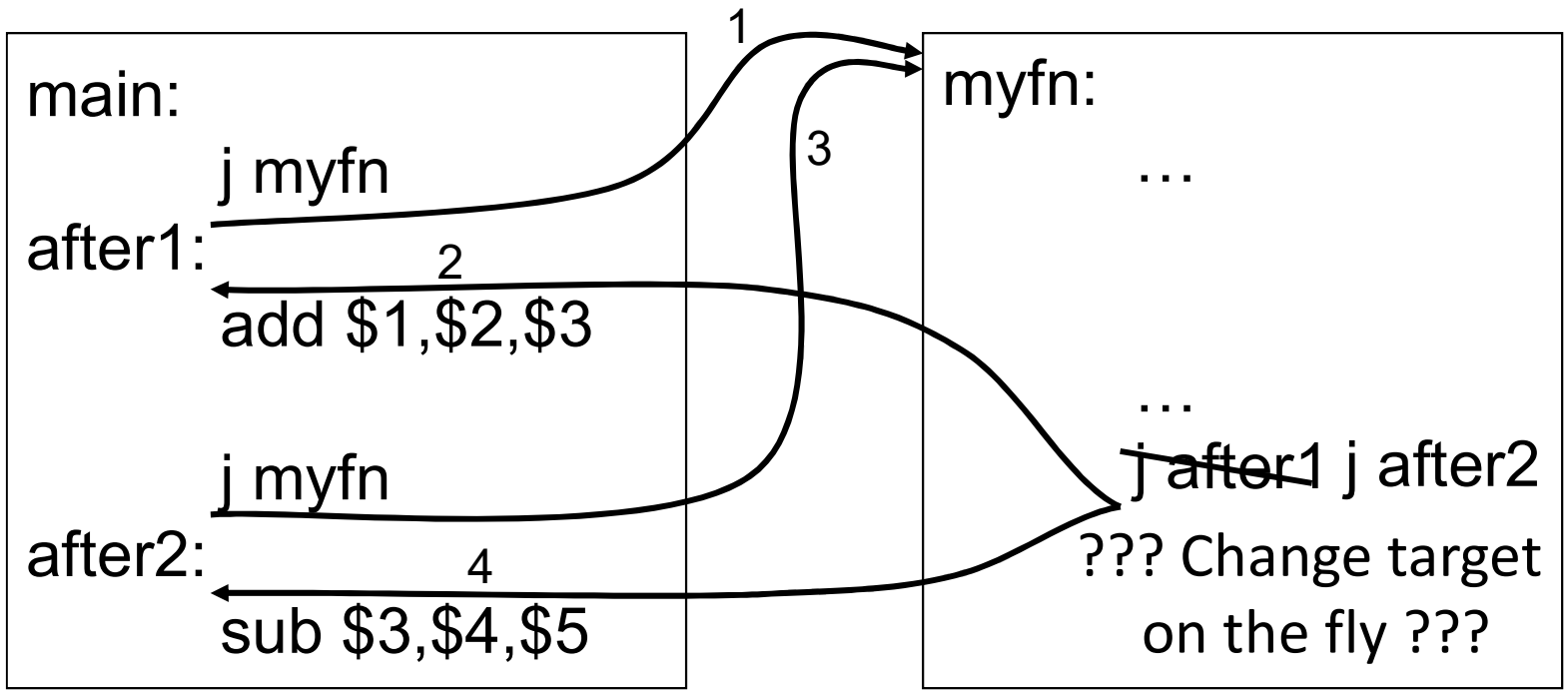
Jumps are not enough



Jumps to the callee

Jumps back

Jumps are not enough



Jumps to the callee

Jumps back

What about multiple sites?

Takeaway1: Need Jump And Link

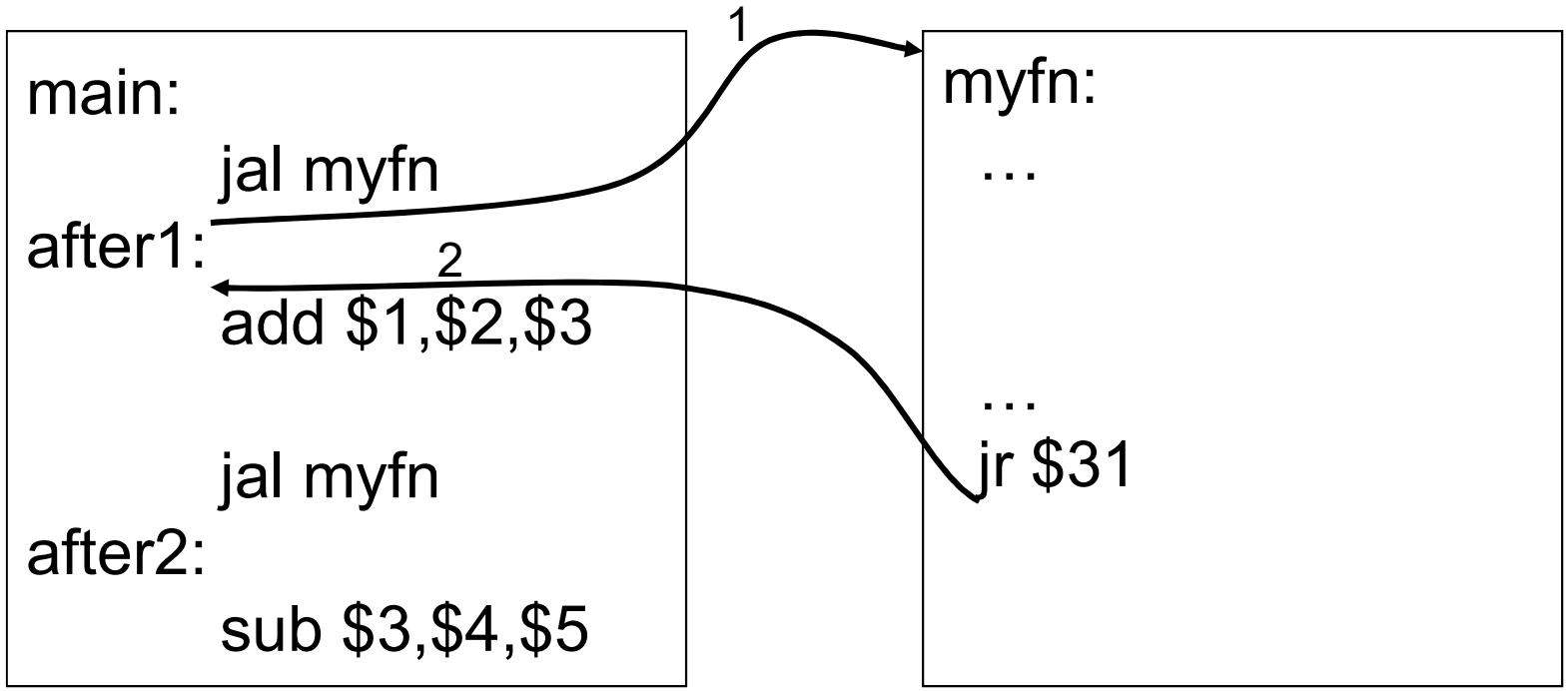
JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register $31 (aka $ra or return address)

Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register $31

Jump-and-Link / Jump Register

First call

r31 | *after1*



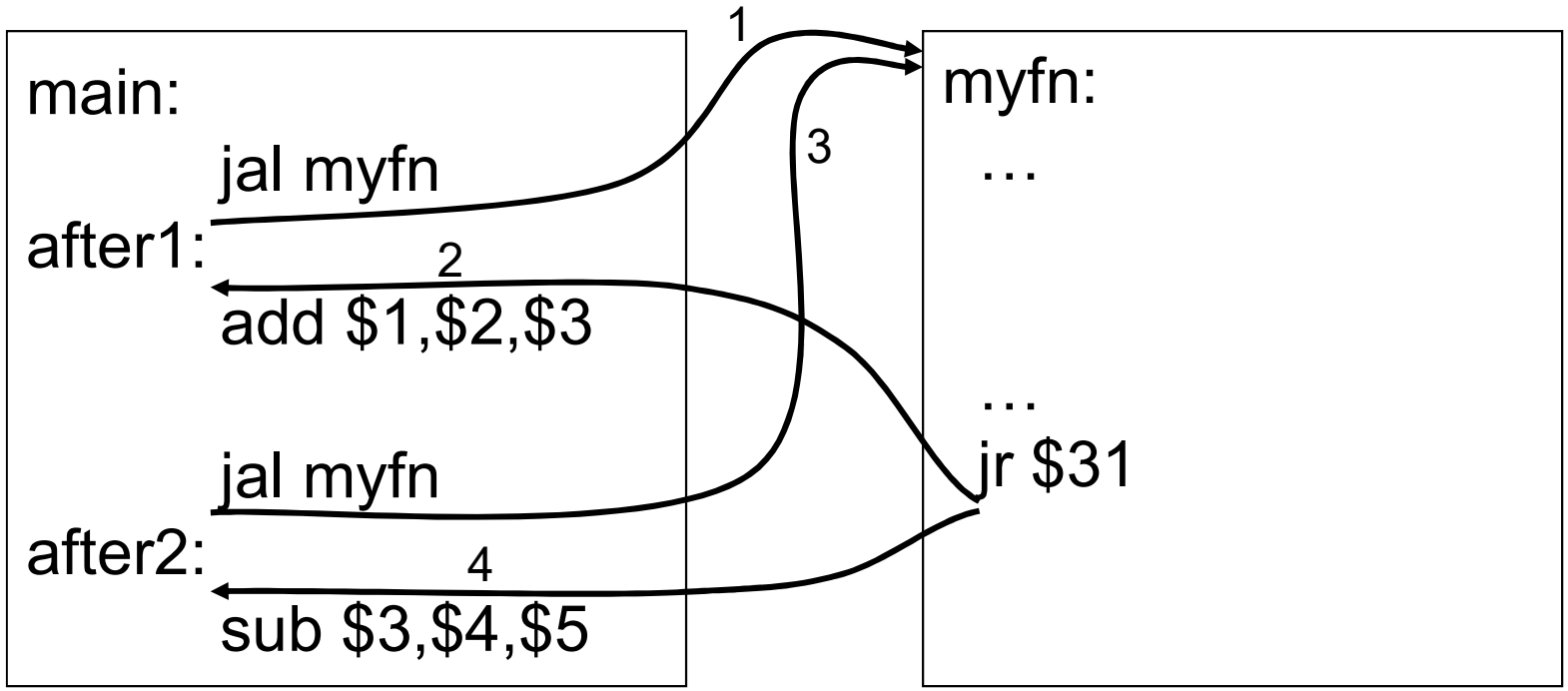
JAL saves the PC in register $31

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

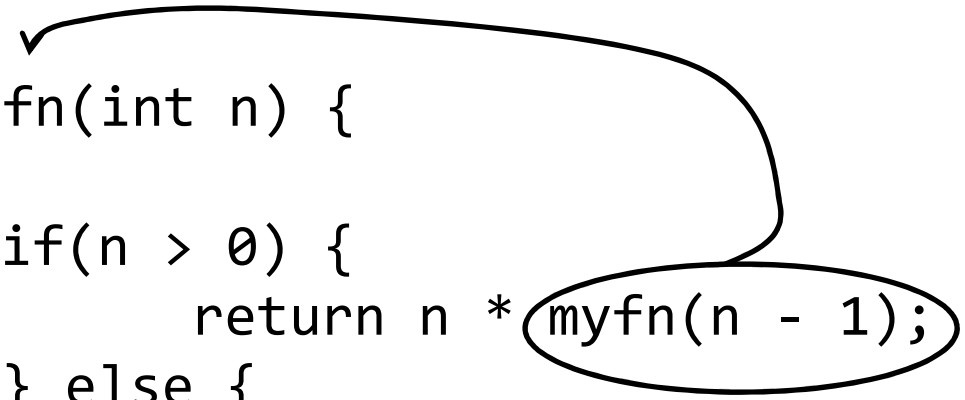
What happens for recursive invocations?

JAL / JR for Recursion?

```
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;
}
```

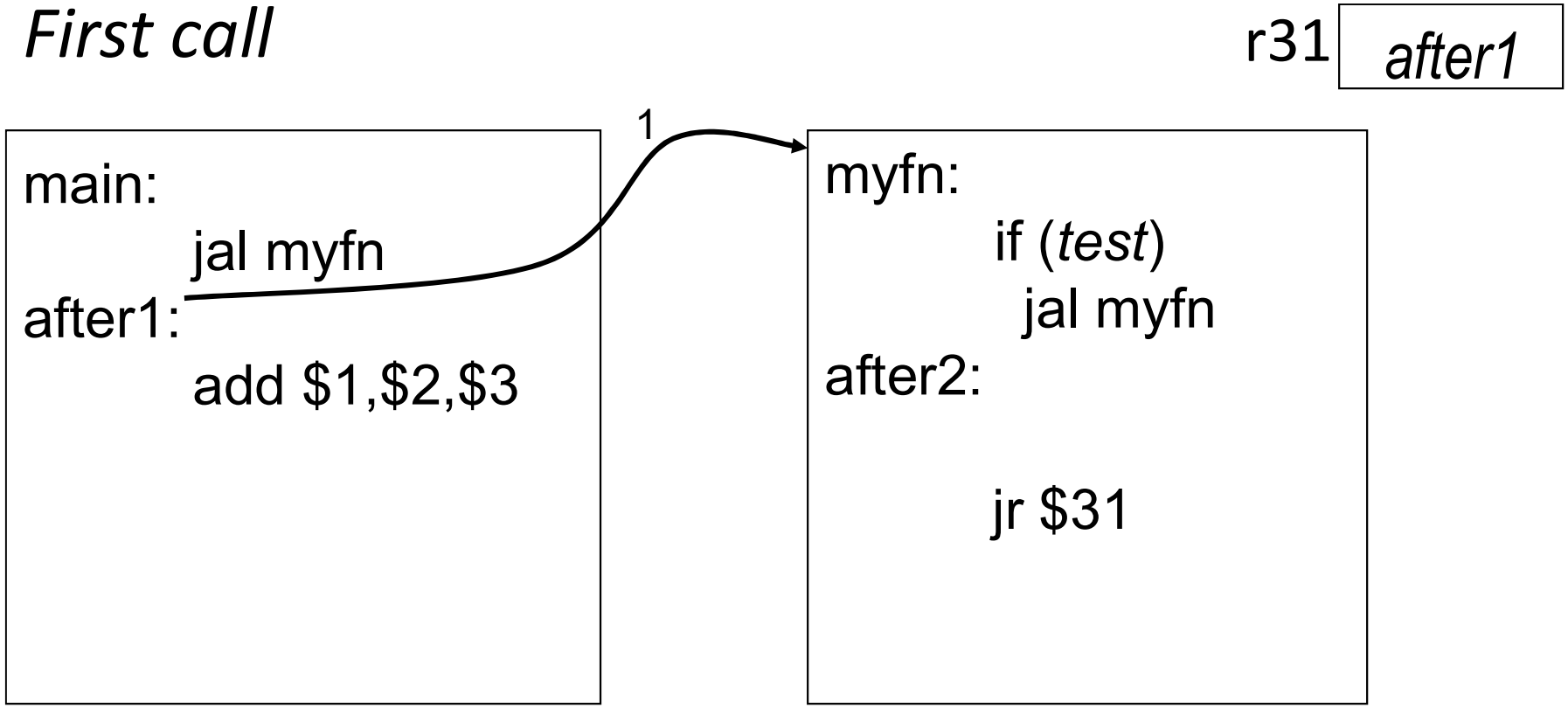
JAL / JR for Recursion?

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

int myfn(int n) {

      if(n > 0) {
            return n * myfn(n - 1);
      } else {
            return 1;
      }
}
```



JAL / JR for Recursion?

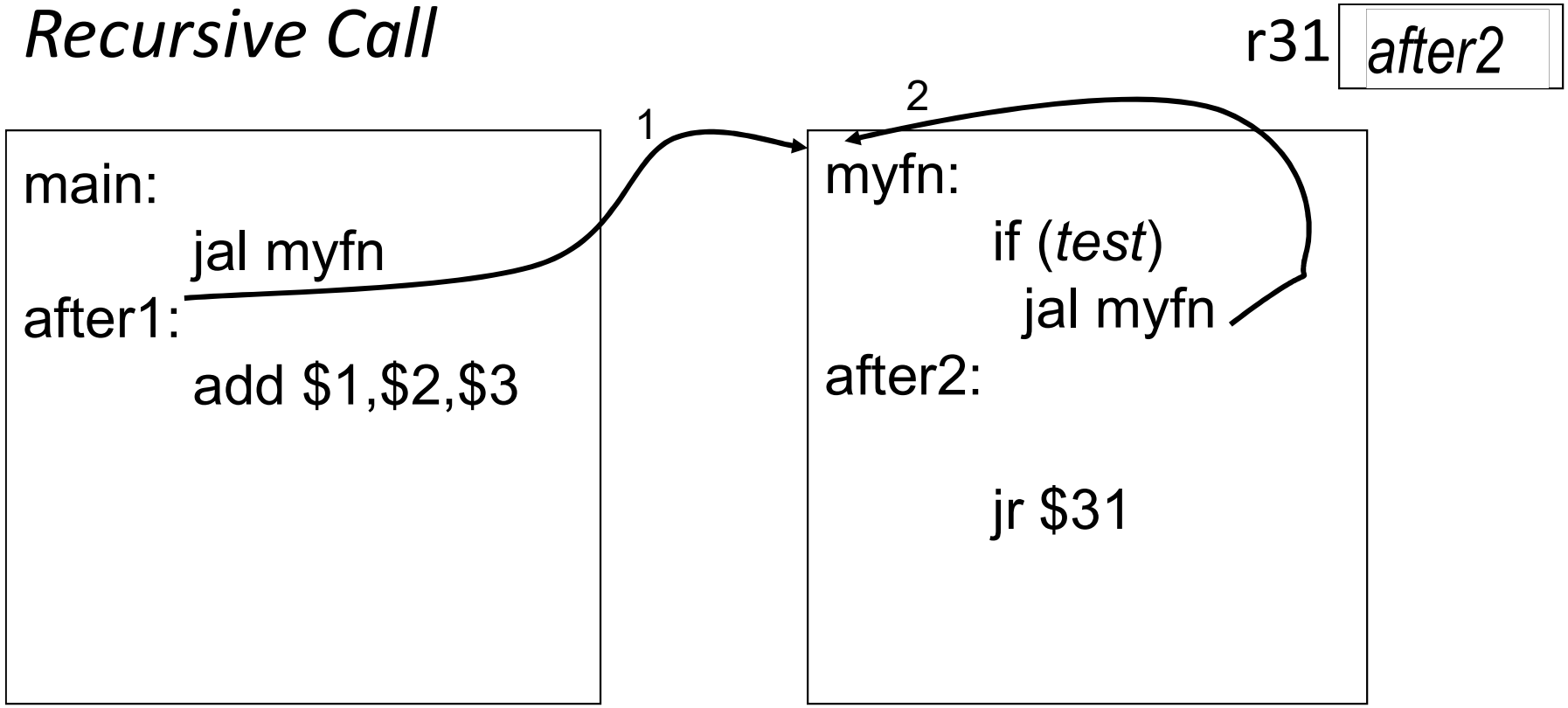
First call



Problems with recursion:

- overwrites contents of $31

JAL / JR for Recursion?

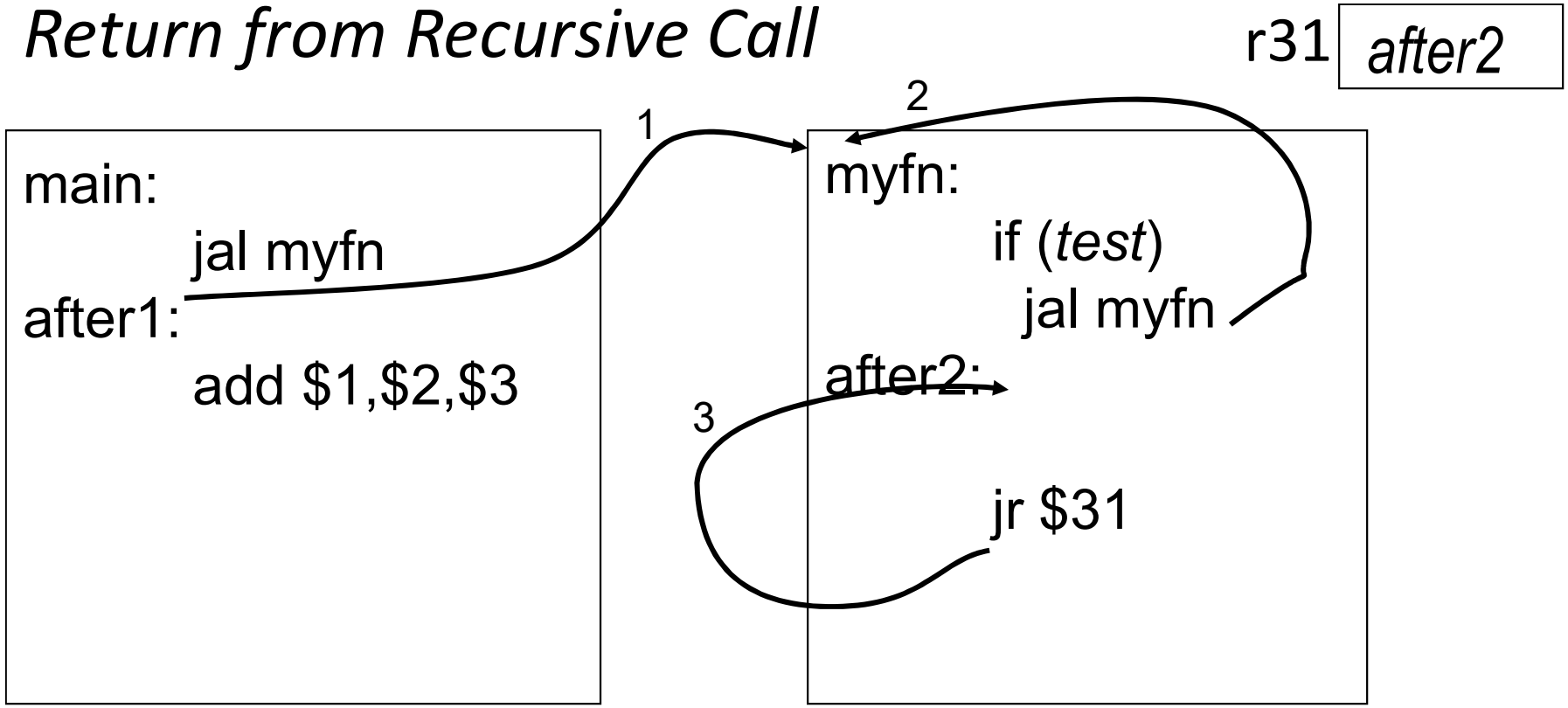


Problems with recursion:

- overwrites contents of $31

JAL / JR for Recursion?

Return from Recursive Call

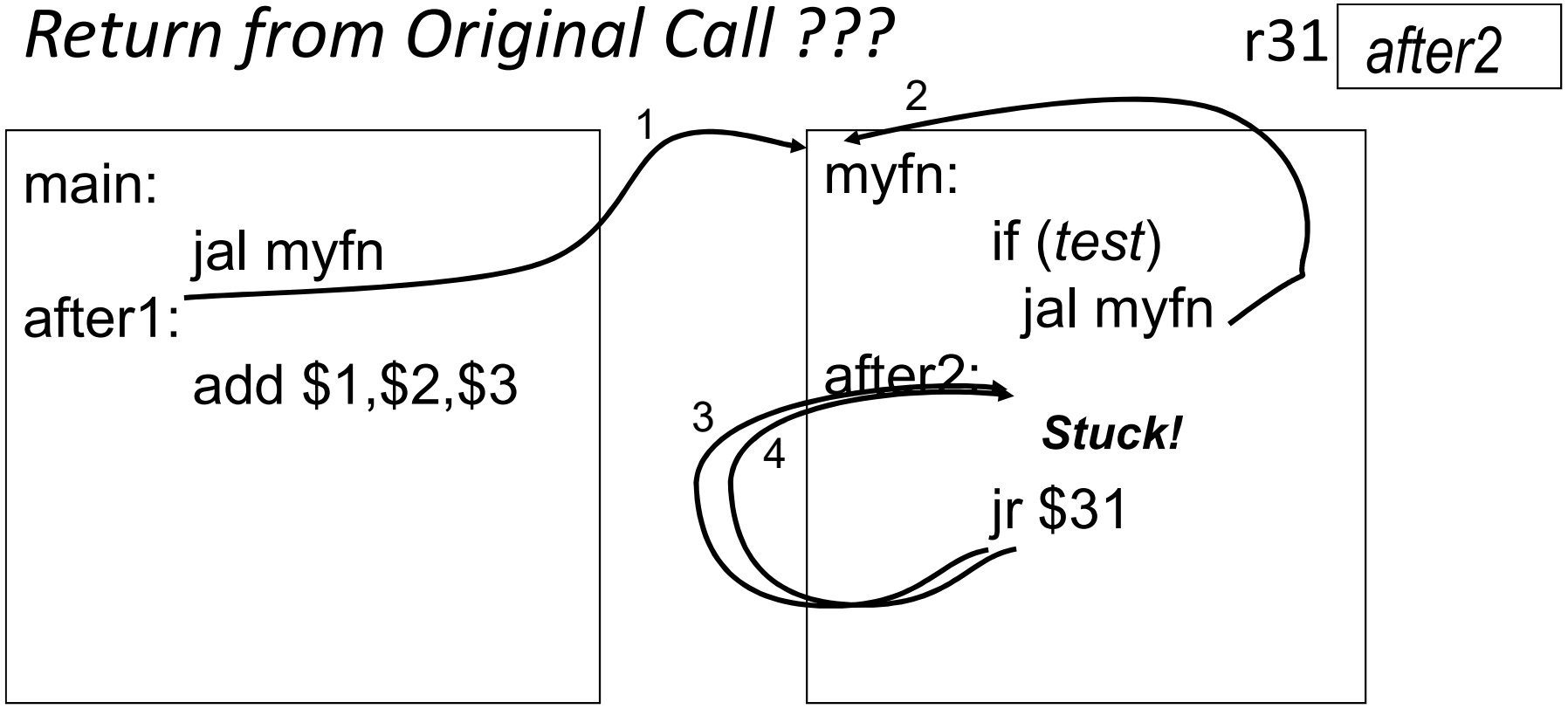


Problems with recursion:

- overwrites contents of $31

JAL / JR for Recursion?

Return from Original Call ???

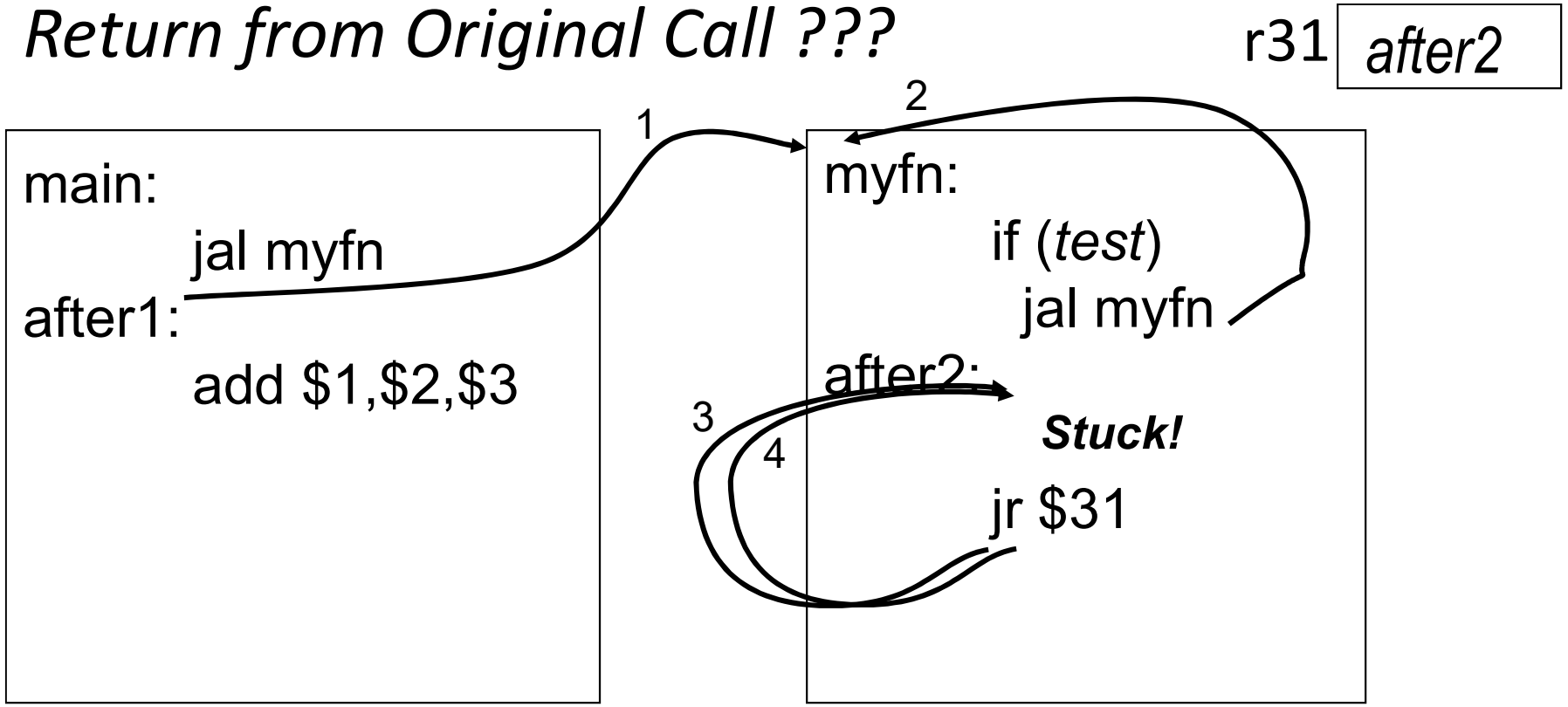


Problems with recursion:

- overwrites contents of $31

JAL / JR for Recursion?

Return from Original Call ???



Problems with recursion:

- overwrites contents of $31
- Need a way to save and restore register contents

Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

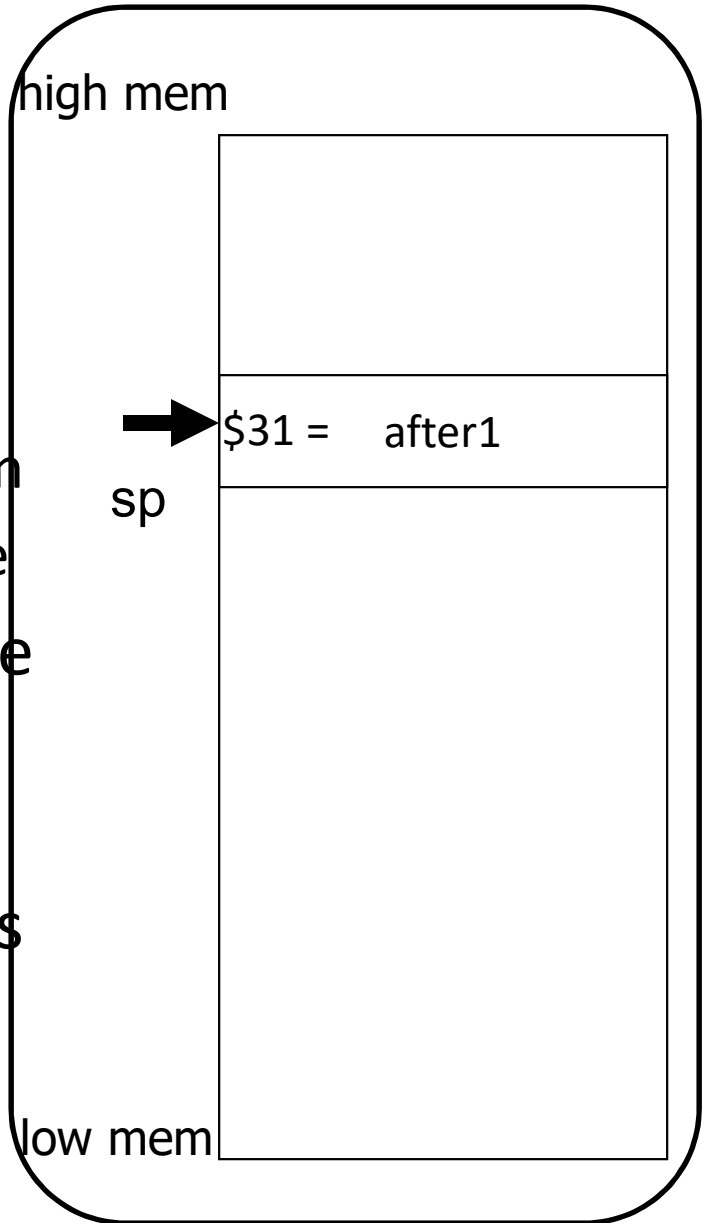
- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack

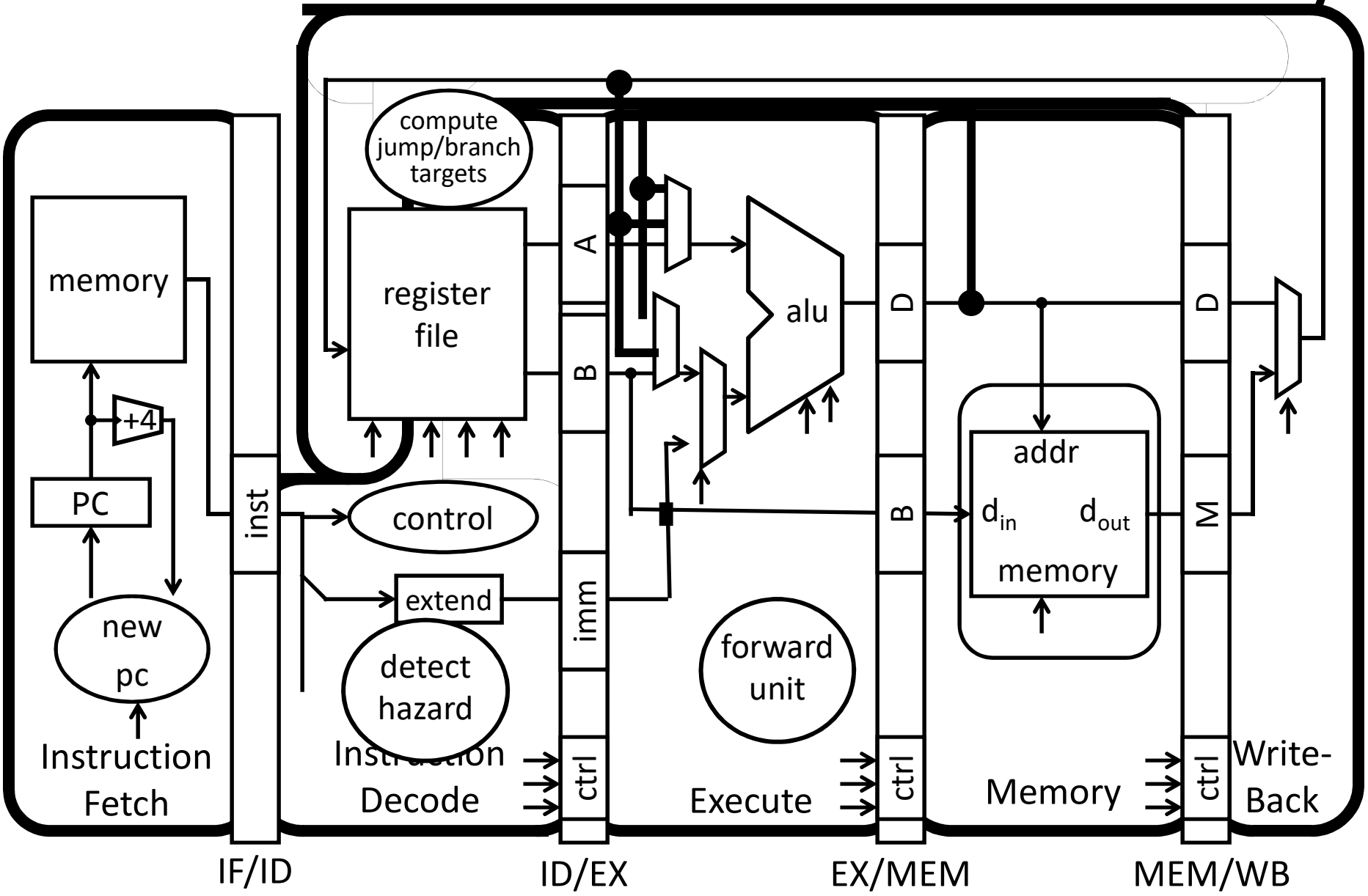
- dedicated register ($29) on the MIPS

Manipulated by push/pop operations

- push: move sp down, store
- pop: load, move sp up



Cheat Sheet and Mental Model for Today



Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

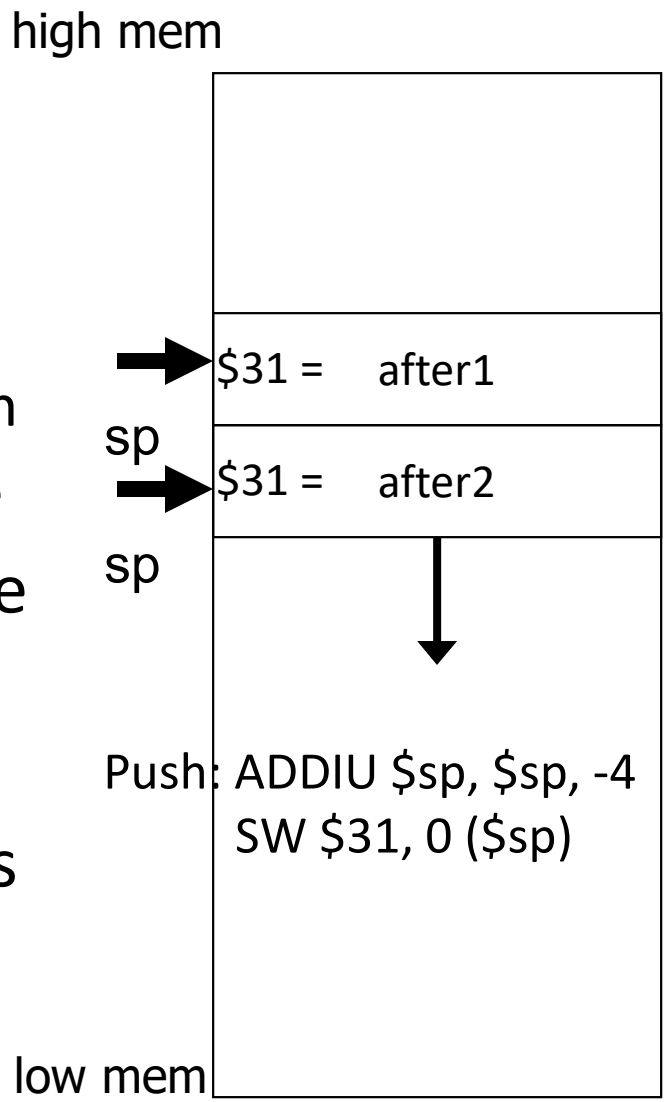
- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack

- dedicated register ($29) on the MIPS

Manipulated by push/pop operations

- push: move sp down, store
- pop: load, move sp up



Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

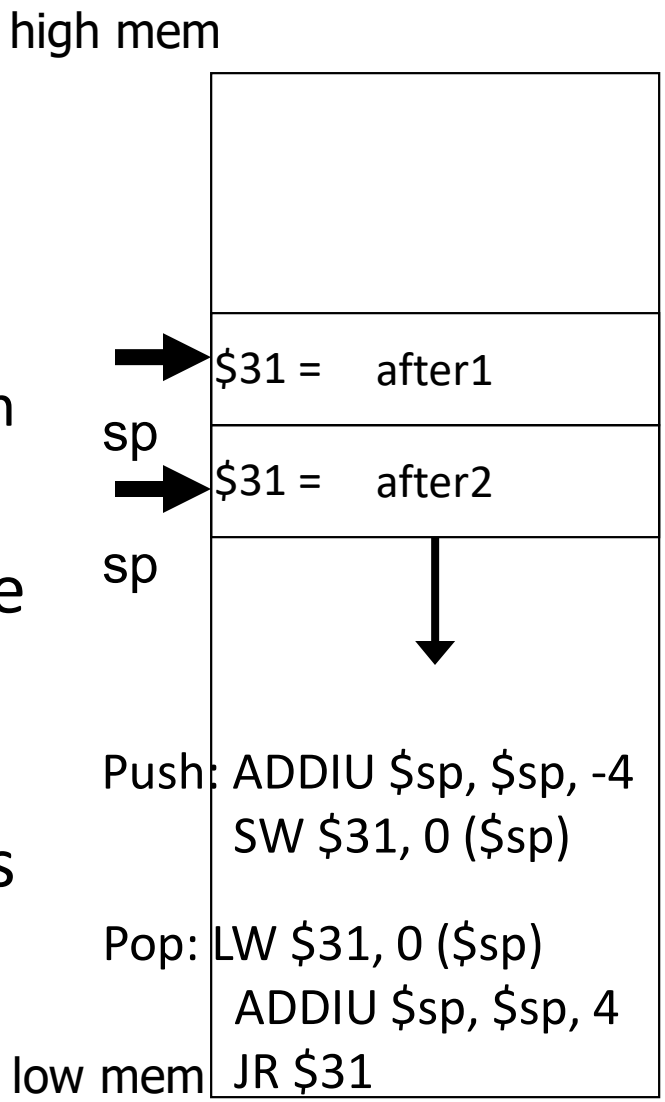
- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack

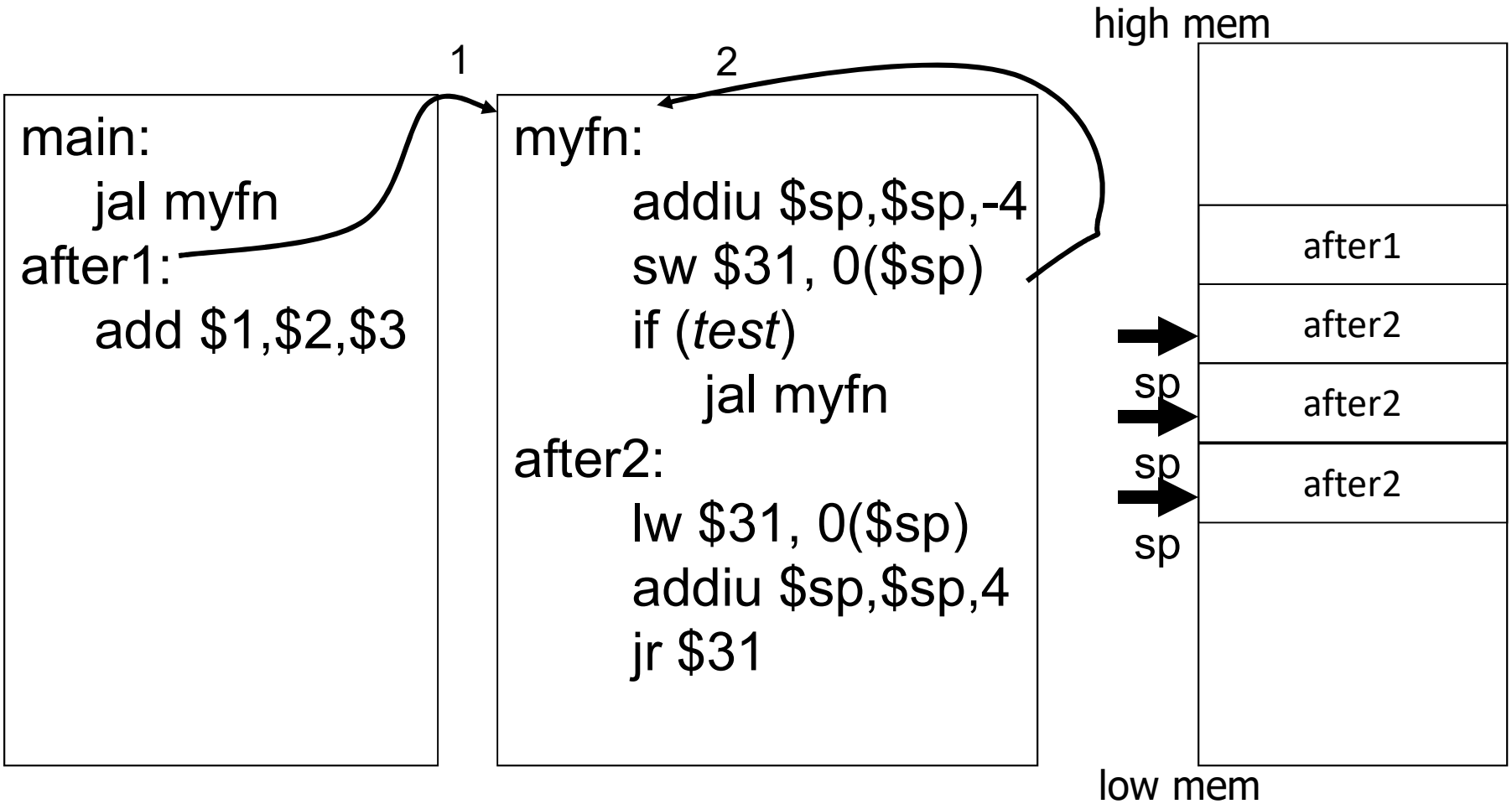
- dedicated register ($29) on the MIPS

Manipulated by push/pop operations

- push: move sp down, store
- pop: load, move sp up

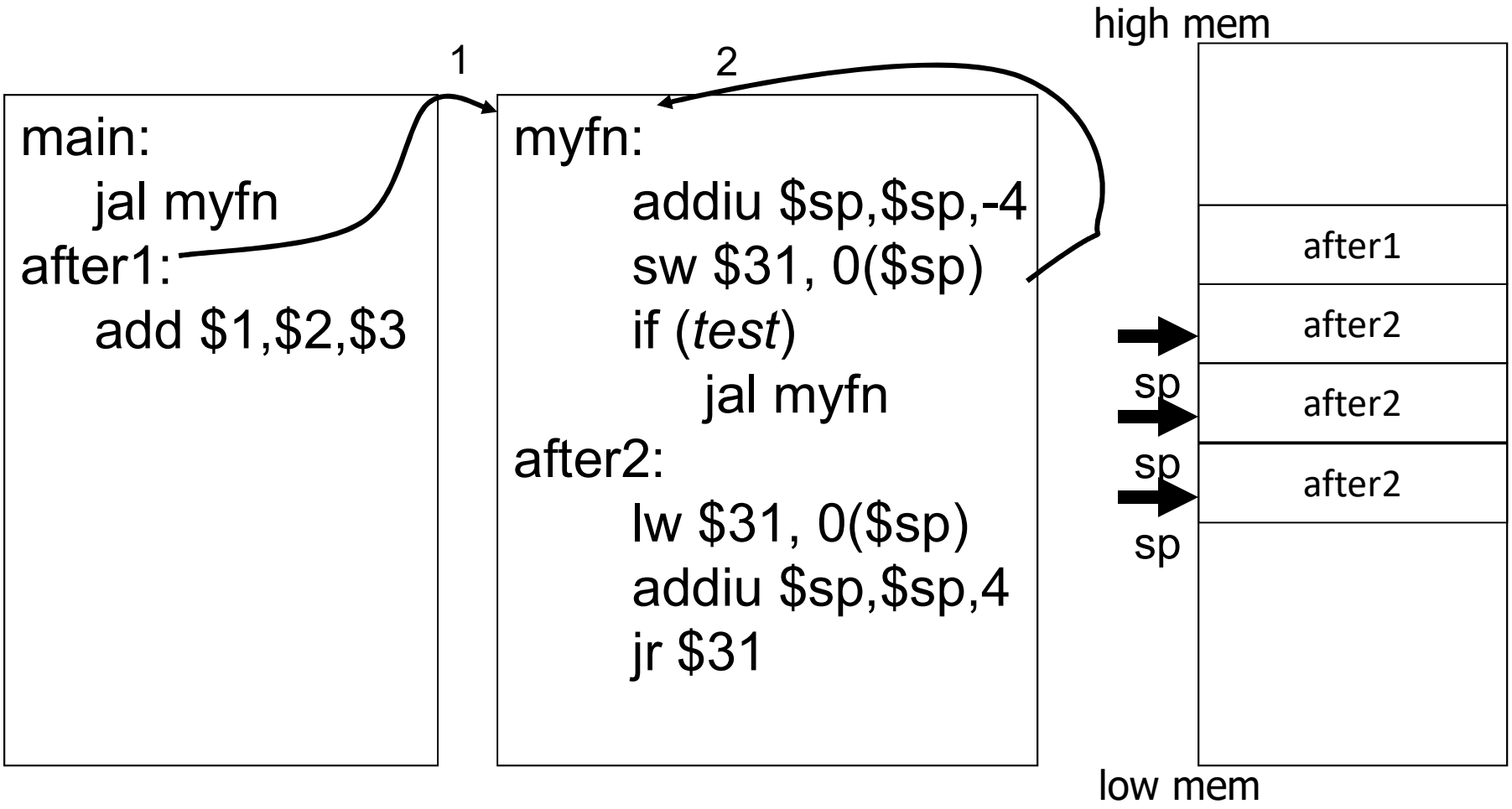


Need a "Call Stack"



Stack used to save and restore contents of $31

Need a "Call Stack"



Stack used to save and restore contents of $31

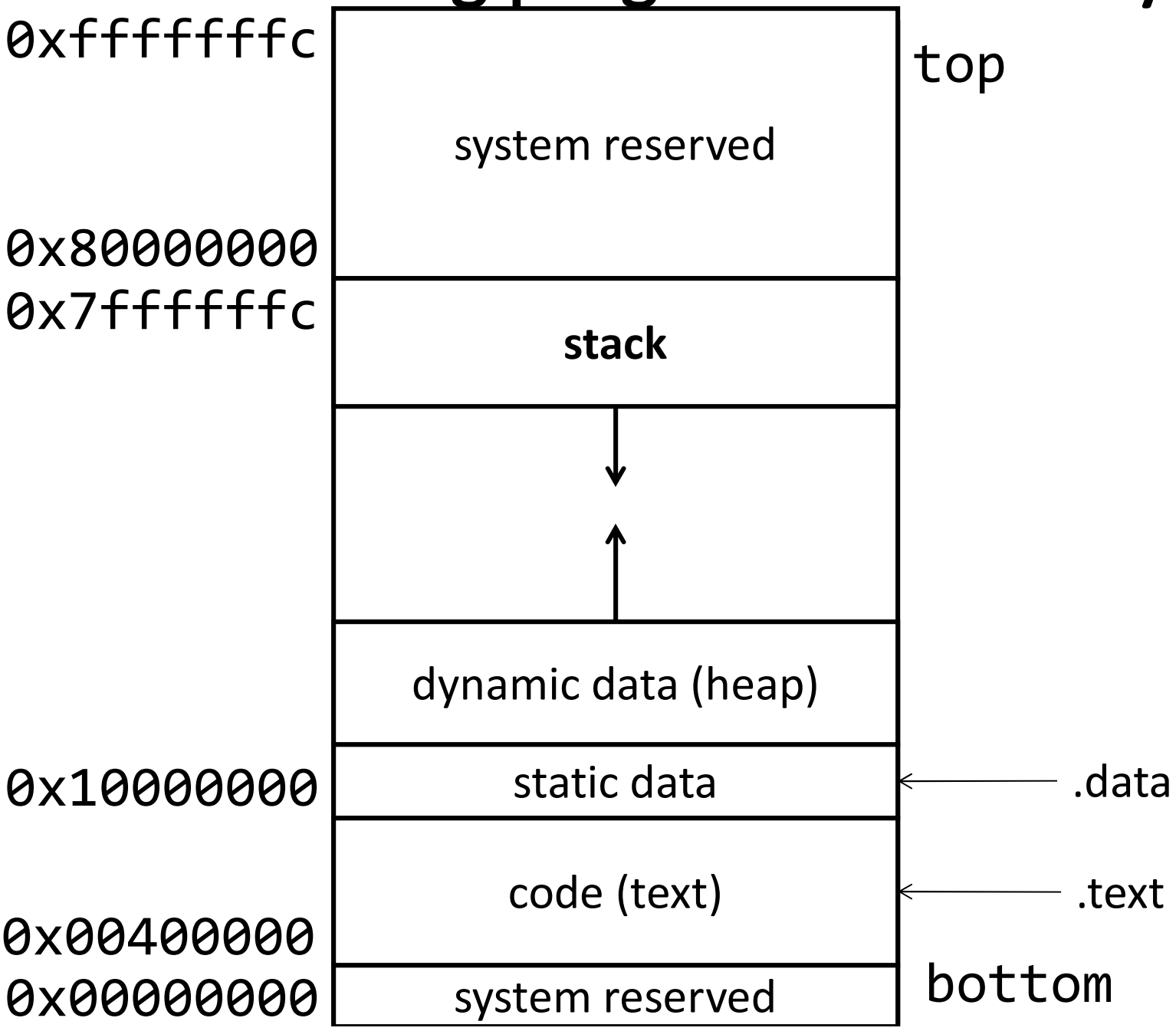
Stack Growth

(Call) Stacks start at a high address in memory

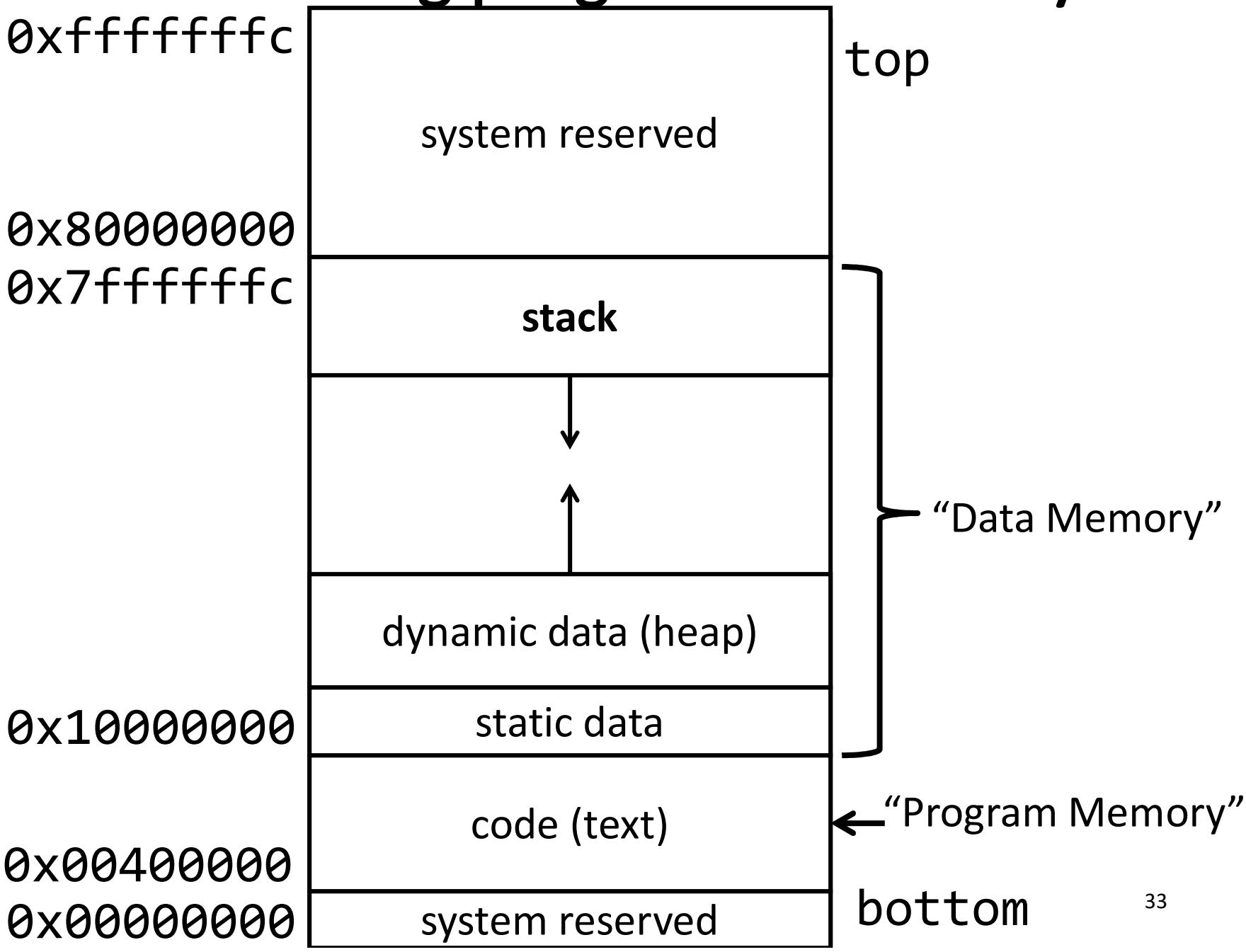
Stacks grow down as frames are pushed on

- Note: data region starts at a low address and grows up
- The growth potential of stacks and data region are not artificially limited

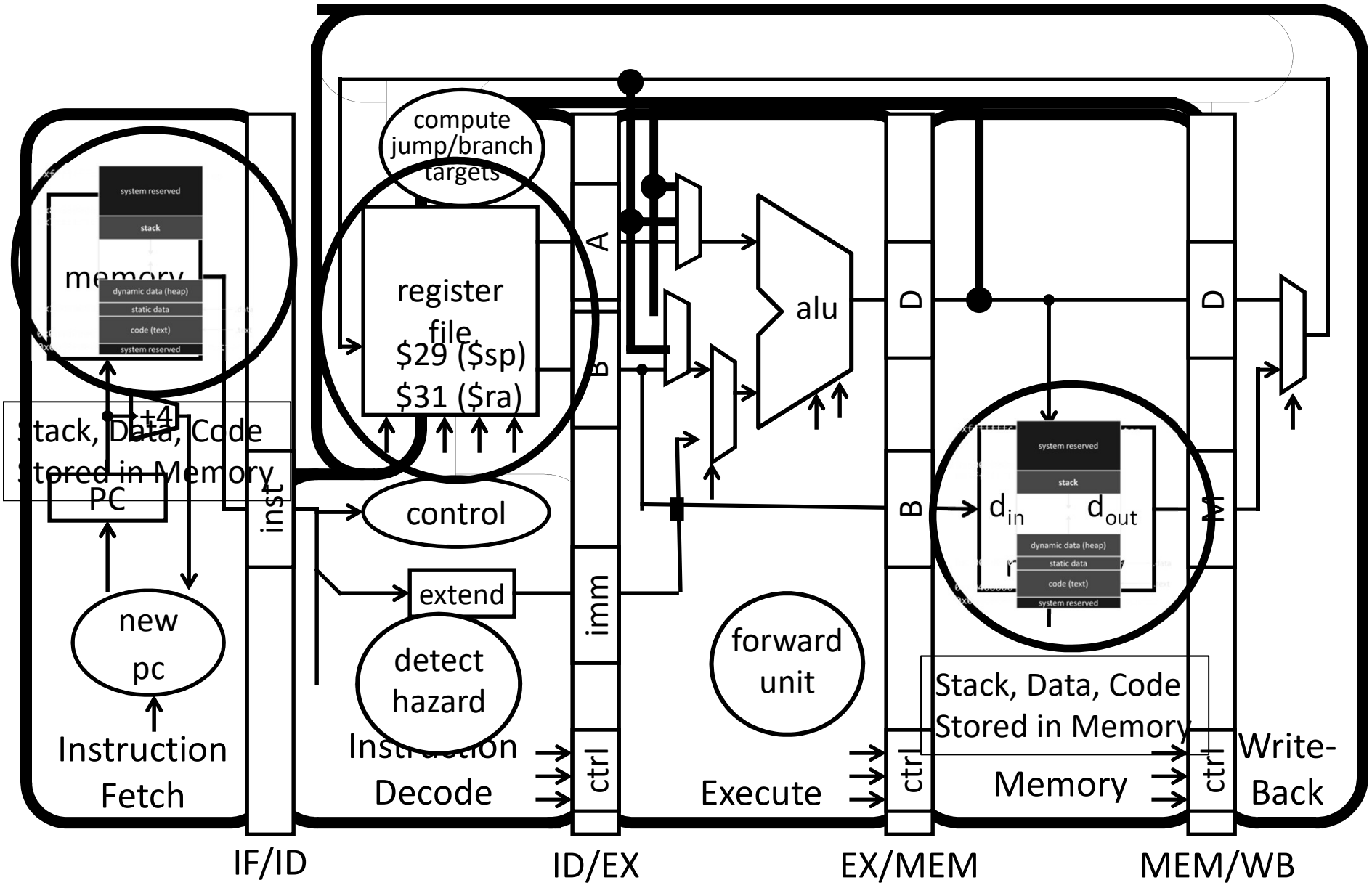
An executing program in memory



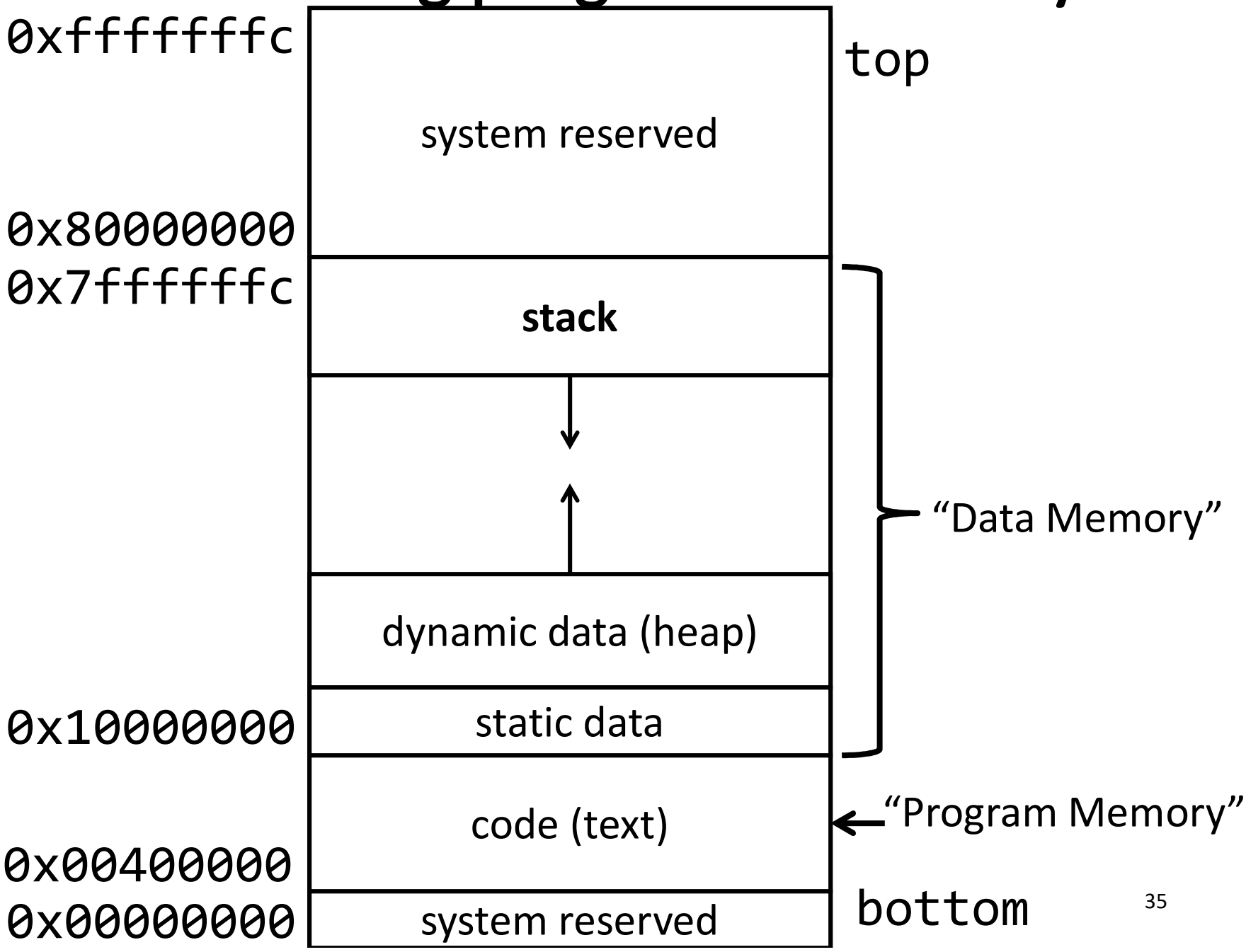
An executing program in memory



Anatomy of an executing program



An executing program in memory



Return Address lives in Stack Frame

Stack Manipulated by push/pop operations

Context: after 2nd JAL to myfn (from myfn)

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

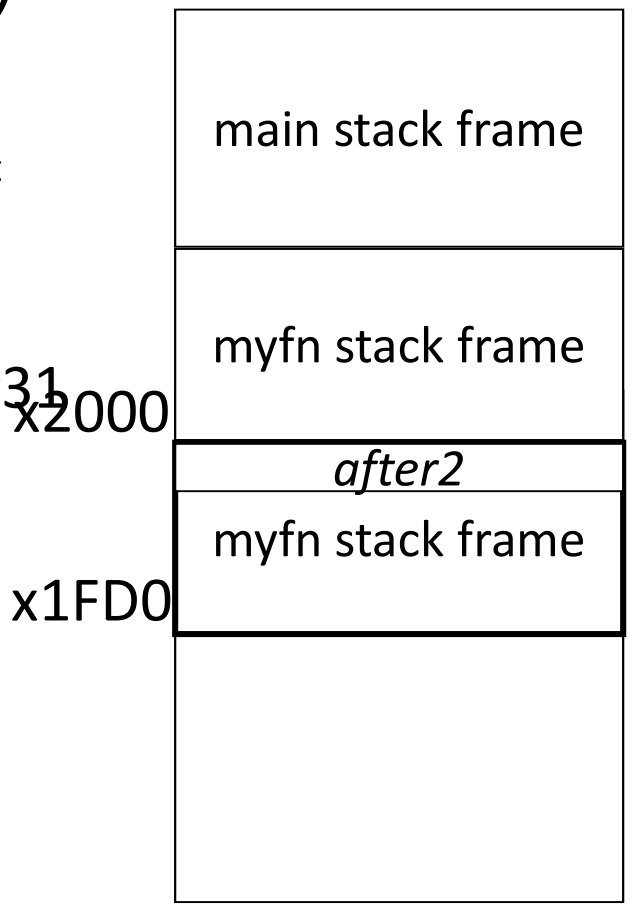
SW $31, 16($sp) // store retn PC 1st

Context: 2nd 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)

The Stack

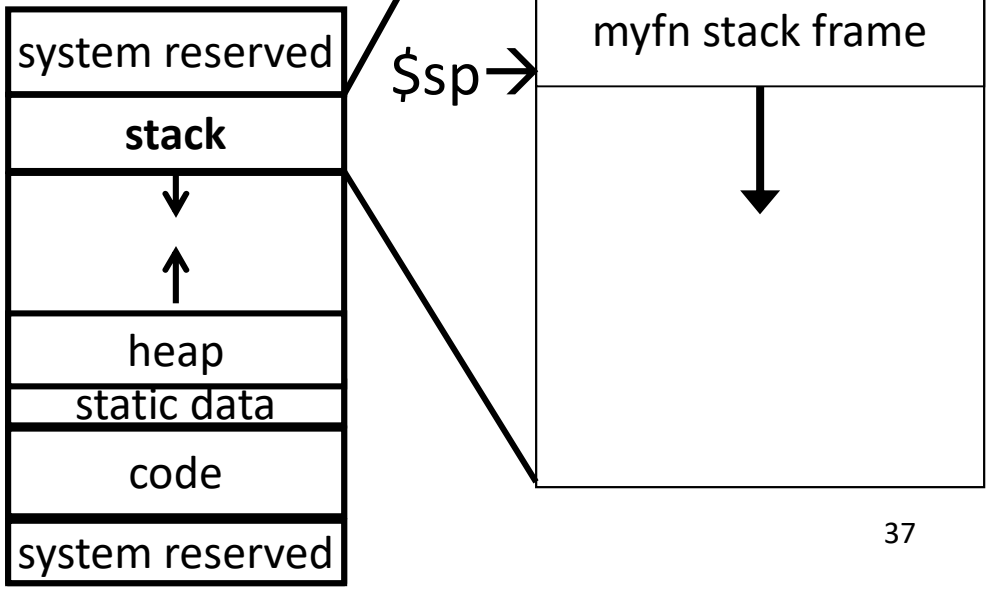
Stack contains stack frames (aka “activation records”)

- 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();
}
```

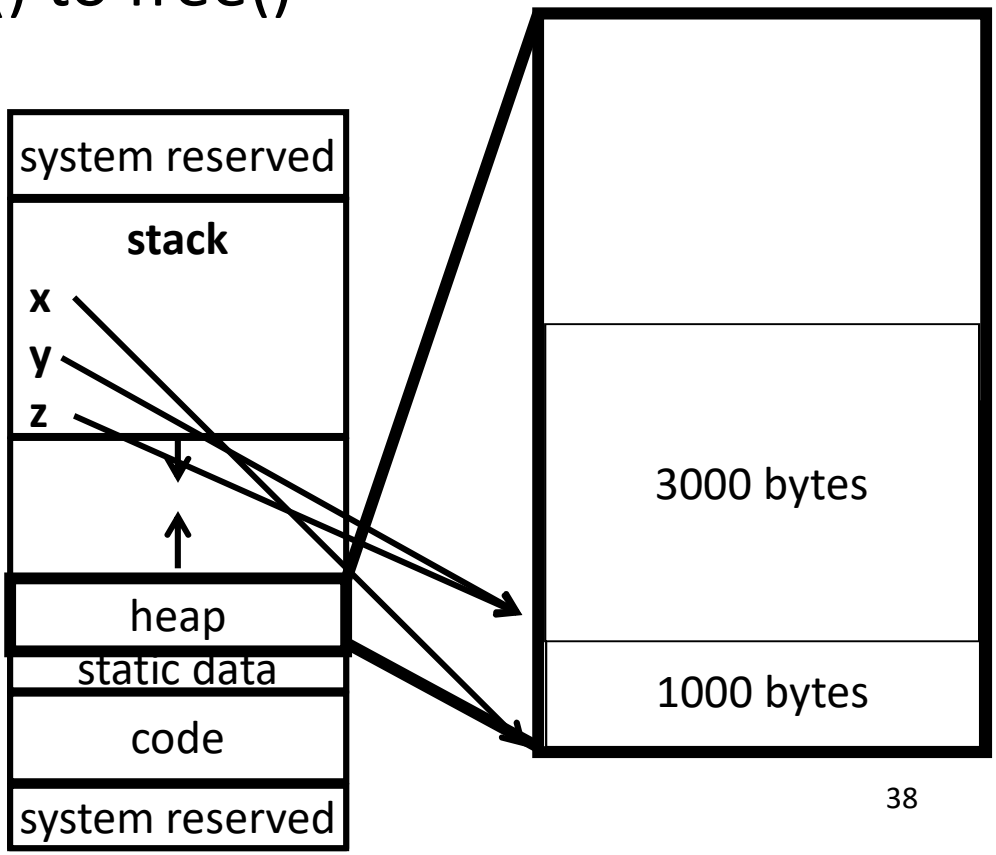


The Heap

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);
}
```

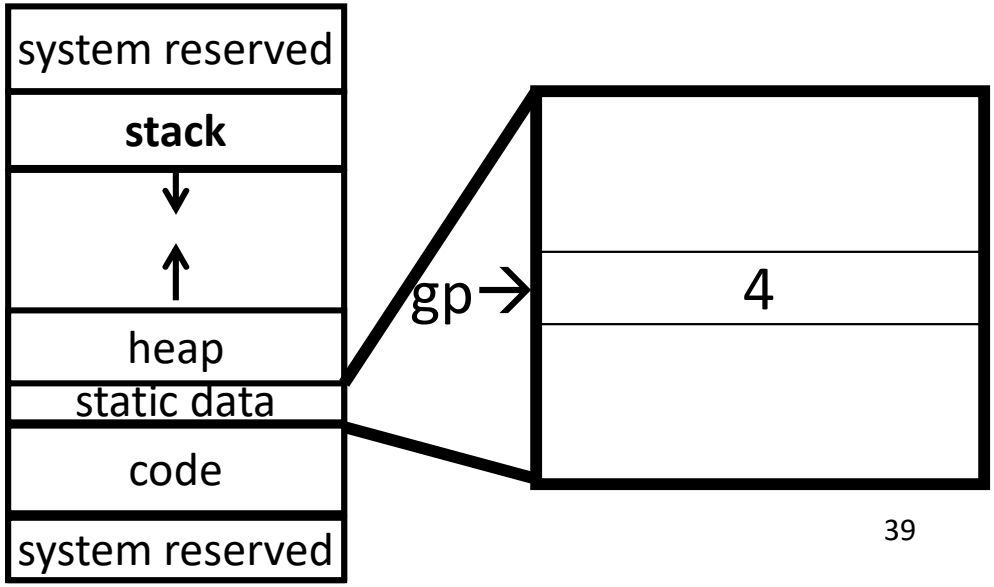


Data Segment

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);
}
```

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);
}
```

Takeaway2: Need a Call Stack

JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register $31 (aka $ra or return address)
Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register $31

Need a Call Stack to return to correct calling procedure. To maintain a stack, need to store an ***activation record*** (aka a "stack frame") in memory. Stacks keep track of the correct return address by storing the contents of $31 in memory (the stack).

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

Next Goal

Need consistent way of passing arguments and getting the result of a subroutine invocation

Arguments & Return Values

Need consistent way of passing arguments and getting the result of a subroutine invocation

Given a procedure signature, need to know where arguments should be placed

- `int min(int a, int b);` — $a0, $a1
- `int subf(int a, int b, int c, int d, int e);` — stack?
- `int isalpha(char c);`
- `int treesort(struct Tree *root);` — $a0
- `struct Node *createNode();` — $v0
- `struct Node mynode();` — $v0, $v1

Too many combinations of char, short, int, void *, struct, etc.

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

Simple Argument Passing (1-4 args)

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

```
main:
   li $a0, 6
   li $a1, 7
   jal myfn
   addiu $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.

Conventions so far:

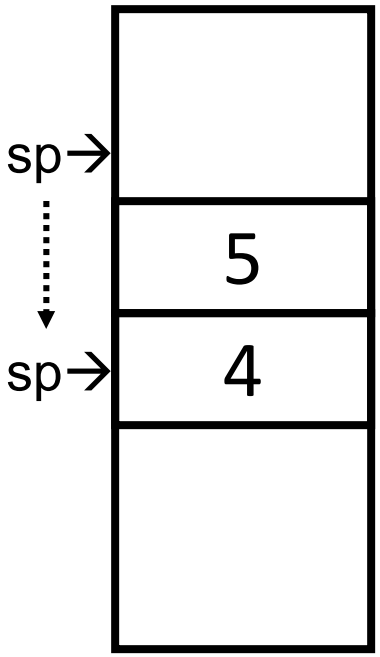
- args passed in $a0, $a1, $a2, $a3
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)

Q: What about argument lists?

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:

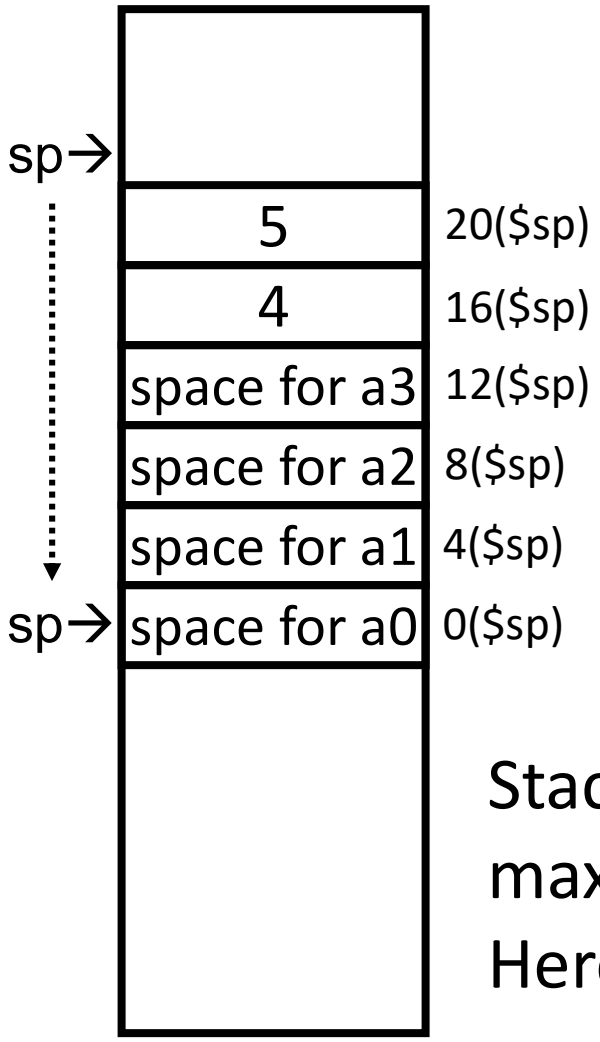
”spill” onto the stack

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, #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.

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);
```

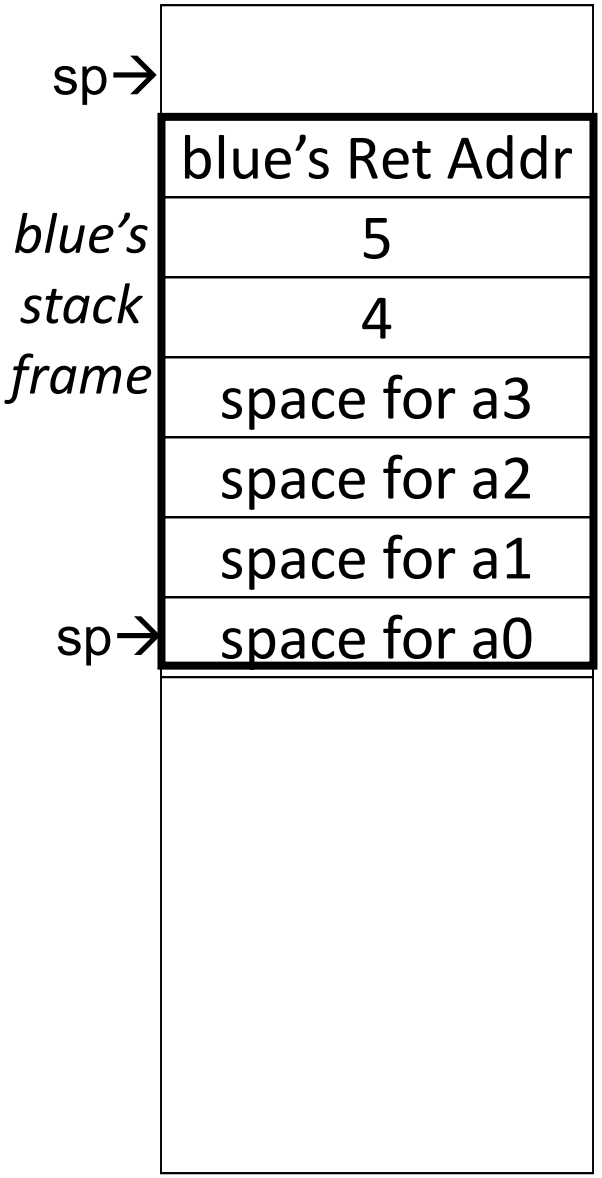
A. Arguments a-e are all passed in registers.

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.

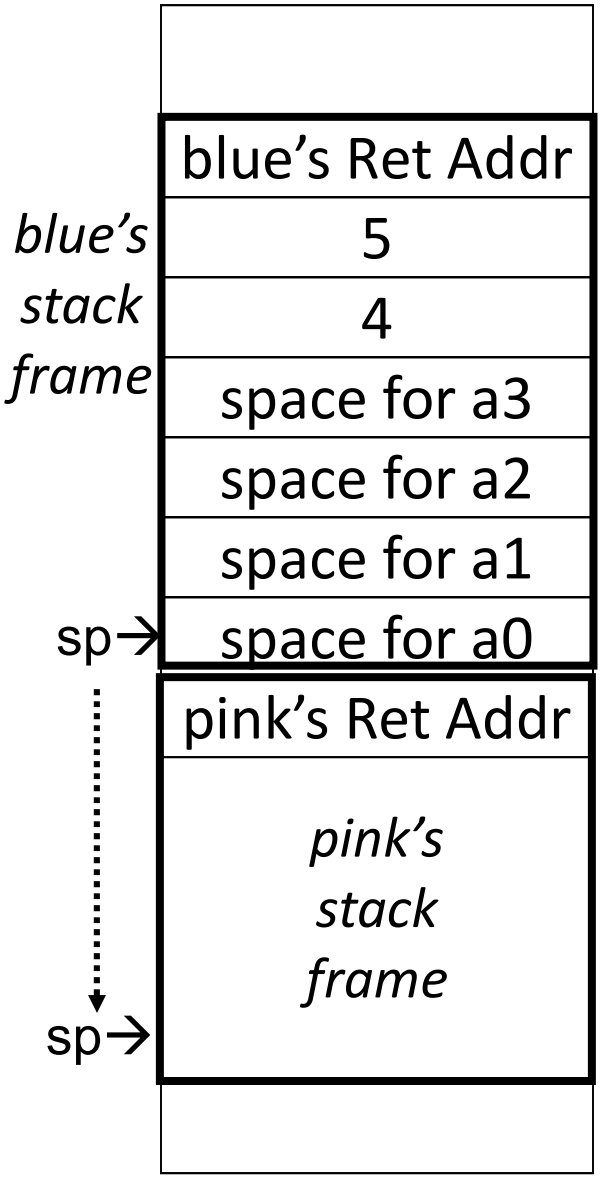
D. Only a-d are stored on the stack, but space is allocated for all 5 arguments.

Frame Layout & the Frame Pointer




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

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) {
    ...
}
```

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, $v1
- stack frame ($fp to $sp) contains:
  - $ra (clobbered on JAL to sub-functions)
  - space for 4 arguments to Callees
  - arguments 5+ to Callees

MIPS Register Conventions so far:

r0	$zero	zero	r16		
r1	$at	assembler temp	r17		
r2	$v0	function	r18		
r3	$v1	return values	r19		
r4	$a0		r20		
r5	$a1	function	r21		
r6	$a2	arguments	r22		
r7	$a3		r23		
r8			r24		
r9			r25		
r10			r26	$k0	reserved
r11			r27	$k1	for OS kernel
r12			r28		
r13			r29		
r14			r30		
r15			r31	$ra	return address

Pseudo-Instructions
e.g. BLZ

SLT $at
BNE $at, 0, L

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);`

$a0, $a1 $a2, $a3

$a0 $a1

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);
```

$a0, $a1

stack

$a0

Where are the return values from:

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

$v0

$v0, $v1

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

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

Globals and Locals

Global variables are allocated in the "data" region of the program

- Exist for all time, accessible to all routines

Local variables are allocated within the stack frame

- Exist solely for the duration of the stack frame

Dangling pointers are pointers into a destroyed stack frame

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

Return the address of a,
But a is stored on stack, so will be removed when call returns and point will be invalid

Global and Locals

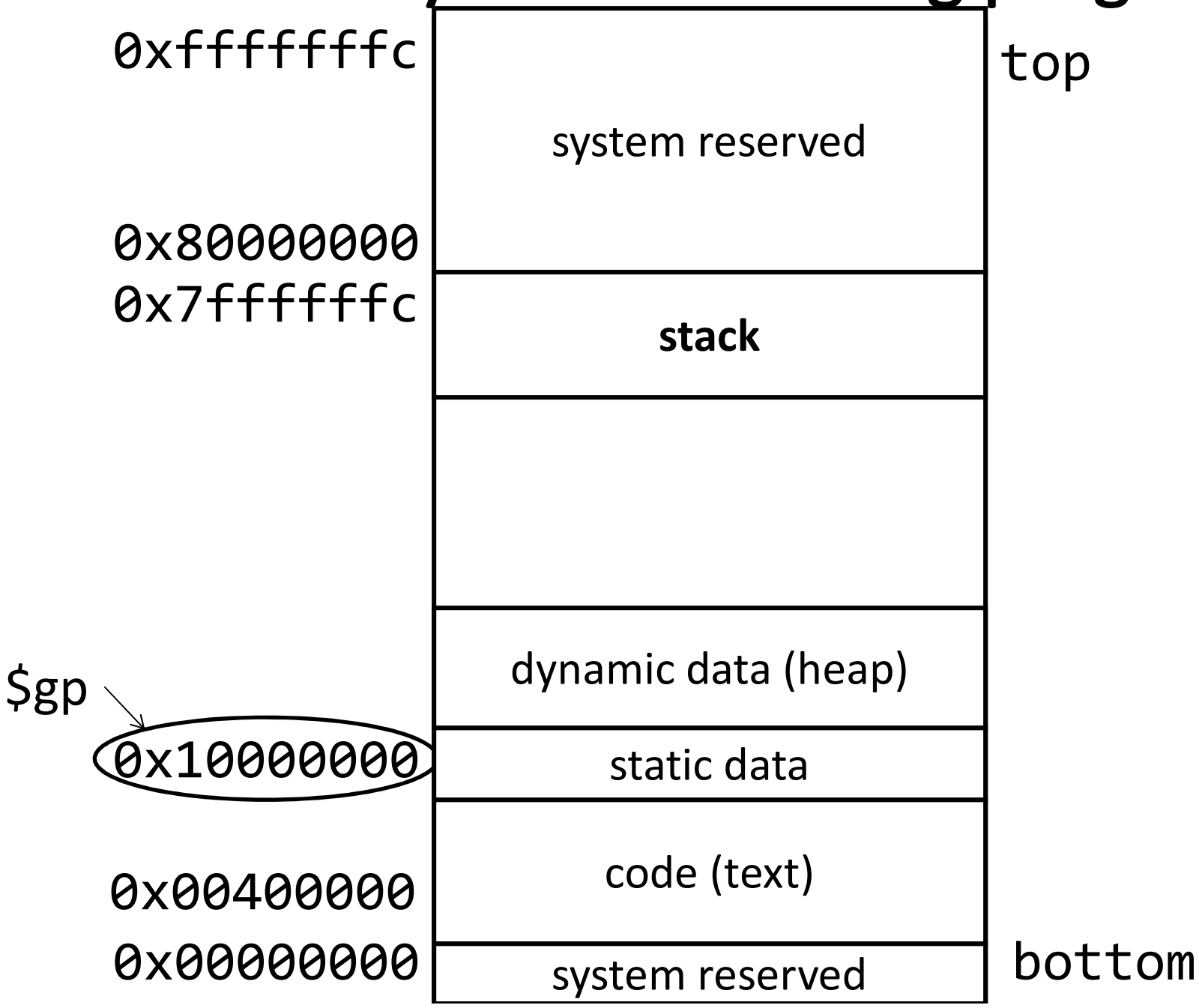
How does a function load global data?

- global variables are just above 0x10000000

Convention: *global pointer*

- $28 is $gp (pointer into *middle* of global data section)
  $gp = 0x10008000
- Access most global data using LW at $gp +/- offset
  LW $v0, 0x8000($gp)
  LW $v1, 0x7FFF($gp)

Anatomy of an executing program



Frame Pointer

It is often cumbersome to keep track of location of data on the stack

- The offsets change as new values are pushed onto and popped off of the stack

Keep a pointer to the bottom of the top stack frame

- Simplifies the task of referring to items on the stack

A frame pointer, $30, aka $fp

- Value of $sp upon procedure entry
- Can be used to restore $sp on exit

Conventions so far

- 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)
  - space for 4 arguments to Callees
  - arguments 5+ to Callees
- global data accessed via $gp

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

Next Goal

What convention should we use to share use of registers across procedure calls?

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

Register Usage

Suppose a routine would like to store a value in a register

Two options: callee-save and caller-save

Callee-save:

- Assume that one of the callers is already using that register to hold a value of interest
- Save the previous contents of the register on procedure entry, restore just before procedure return
- E.g. $31

Caller-save:

- Assume that a caller can clobber any one of the registers
- Save the previous contents of the register before proc call
- Restore after the call

MIPS calling convention supports both

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

Where do we save and restore?

Functions

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

```
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 $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, 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.

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 $31,28($sp)
  sw $30, 24($sp)
  sw $17, 20($sp)
  sw $16, 16($sp)
  addiu  $fp, $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

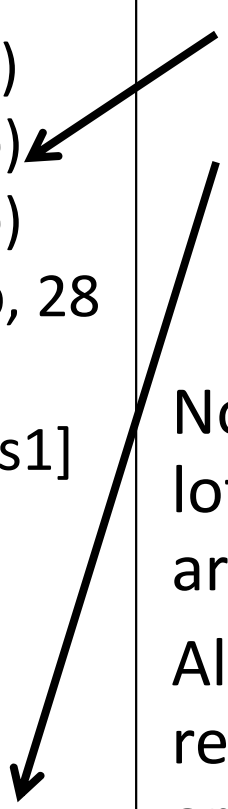
- Save on entry
- Restore on exit

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.

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.

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

Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to 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 valid

.

Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to 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 valid

.

Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to 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 valid

Repeat but assume that foo is recursive (bar/baz → foo)

Clicker Question

```
int foo() {
    int a = 0;
    int b = 12;
    int c = 1;

    while(b + c > 0) {
        int e = b + bar(c);
        c = b + e;

        int d = c + baz(b);
        a = d - e;
    }

    return a;
}
```

You are a compiler. Do you choose to put a in a:

(A) Caller-saved register (t)

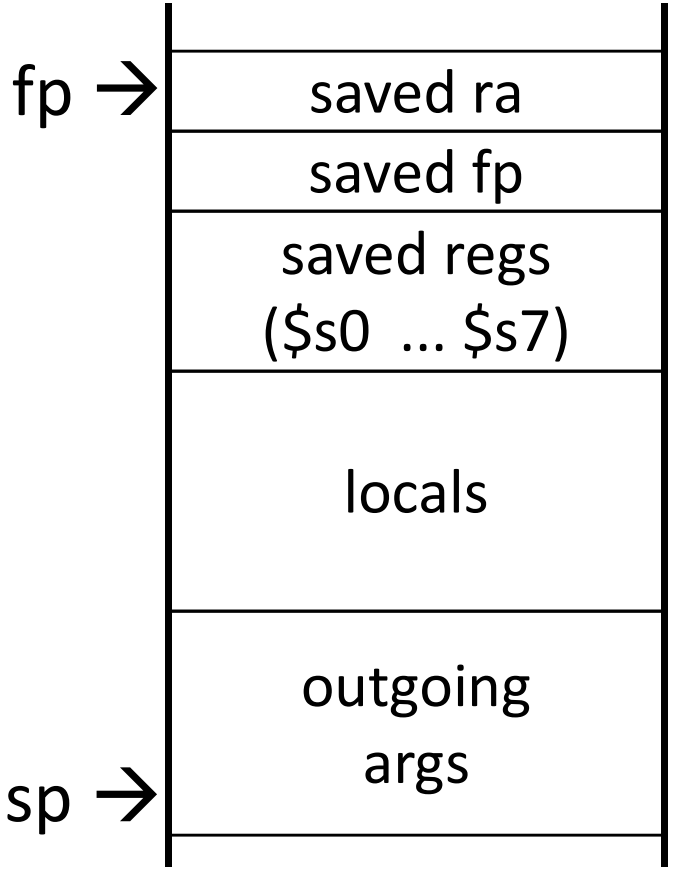
(B) Callee-saved register (s)

(C) Depends on where we put the other variables in this fn

(D) Both are equally valid

Repeat but assume that foo is recursive (bar/baz → foo)

Frame Layout on Stack



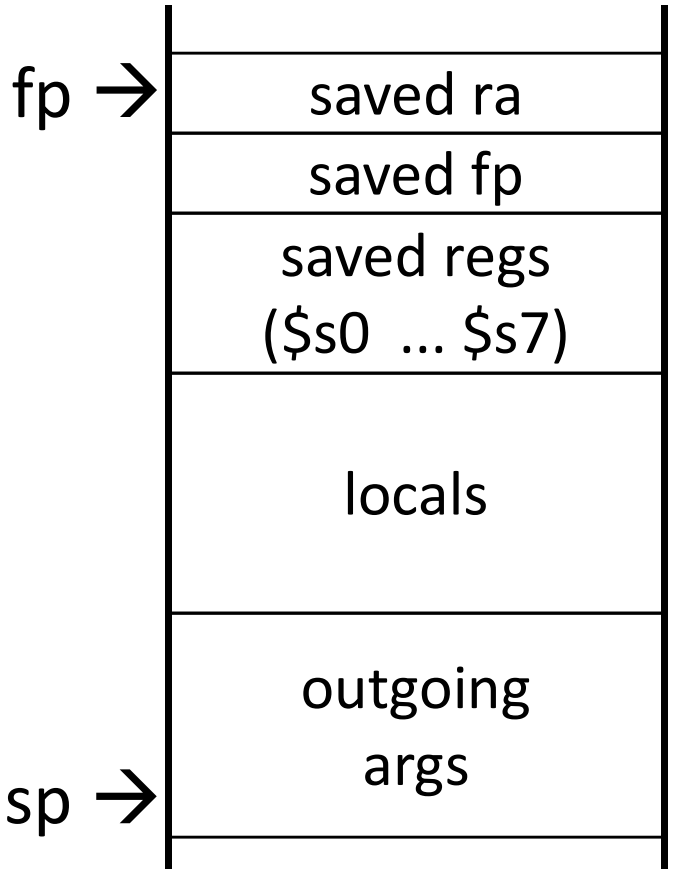
Assume a function uses two callee-save registers.

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

What should be stored in the stack frame?

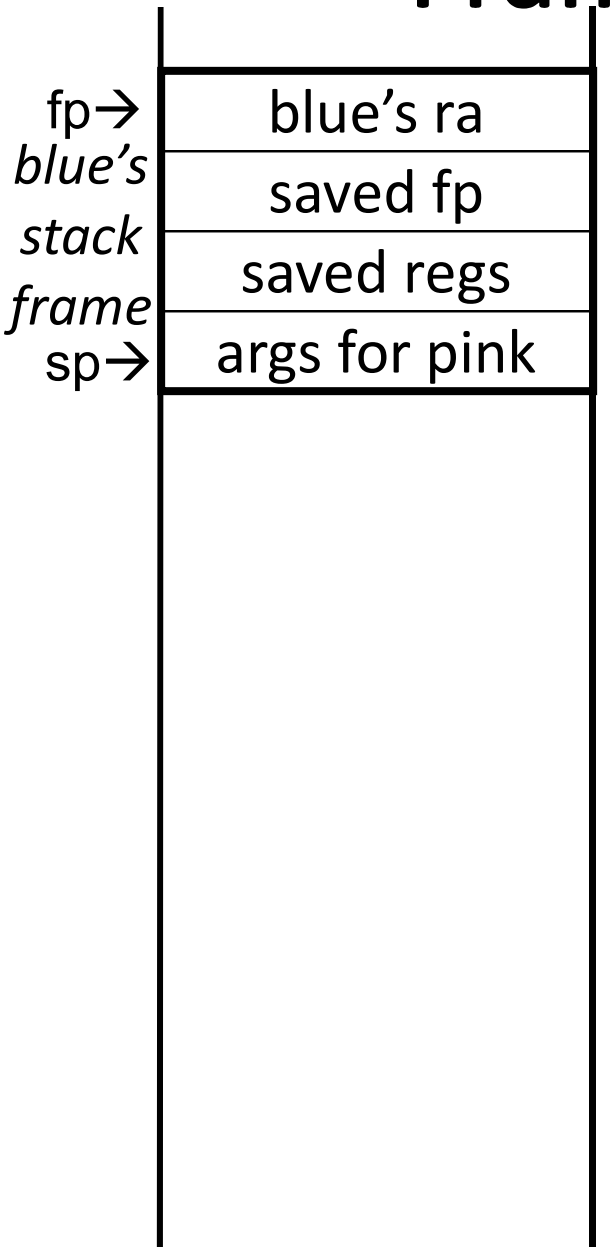
Where should everything be stored?

Frame Layout on Stack



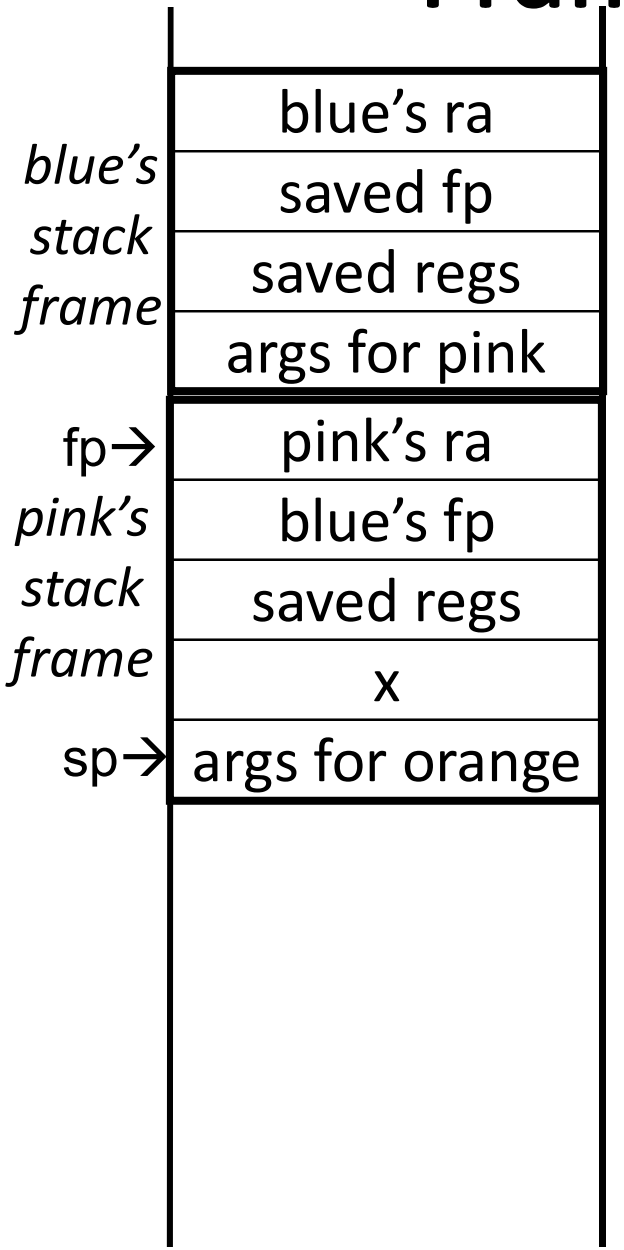

```
ADDIU $sp, $sp, -32     # allocate frame
SW $ra, 28($sp)         # save $ra
SW $fp, 24($sp)         # save old $fp
SW $s1, 20($sp)         # save ...
SW $s0, 16($sp)         # save ...
ADDIU $fp, $sp, 28      # set new frame ptr
…                       ...
BODY
…                       ...
LW $s0,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



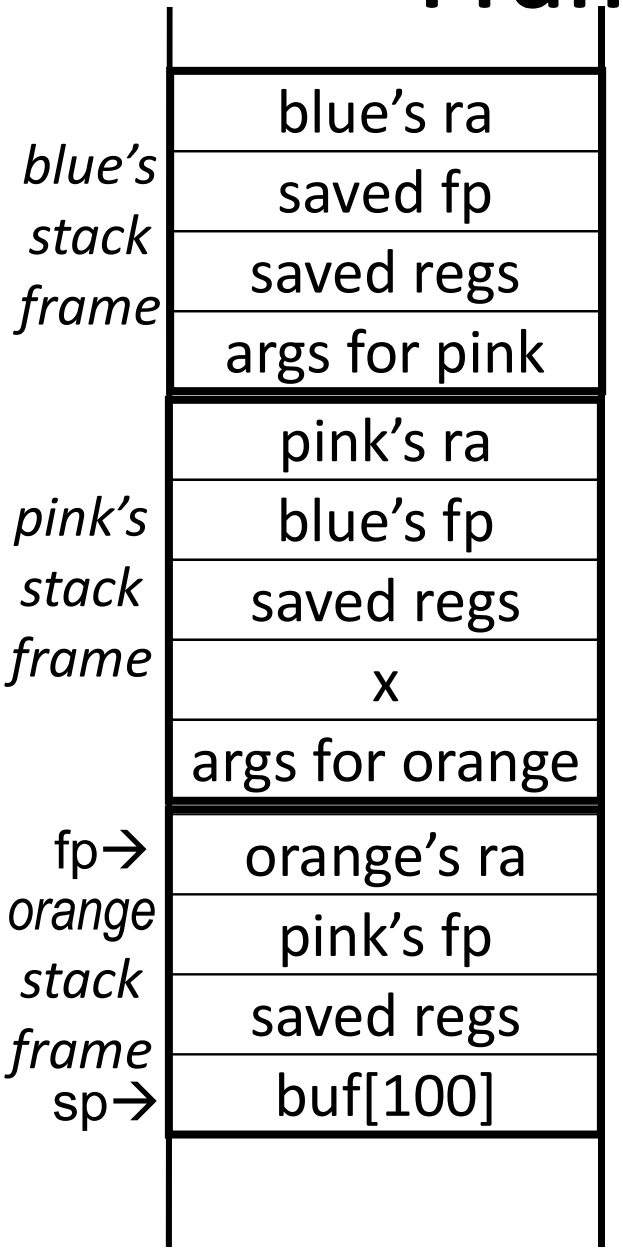

```
blue() {
   pink(0,1,2,3,4,5);
}
```

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) {
    int x;
    orange(10,11,12,13,14);
}
```

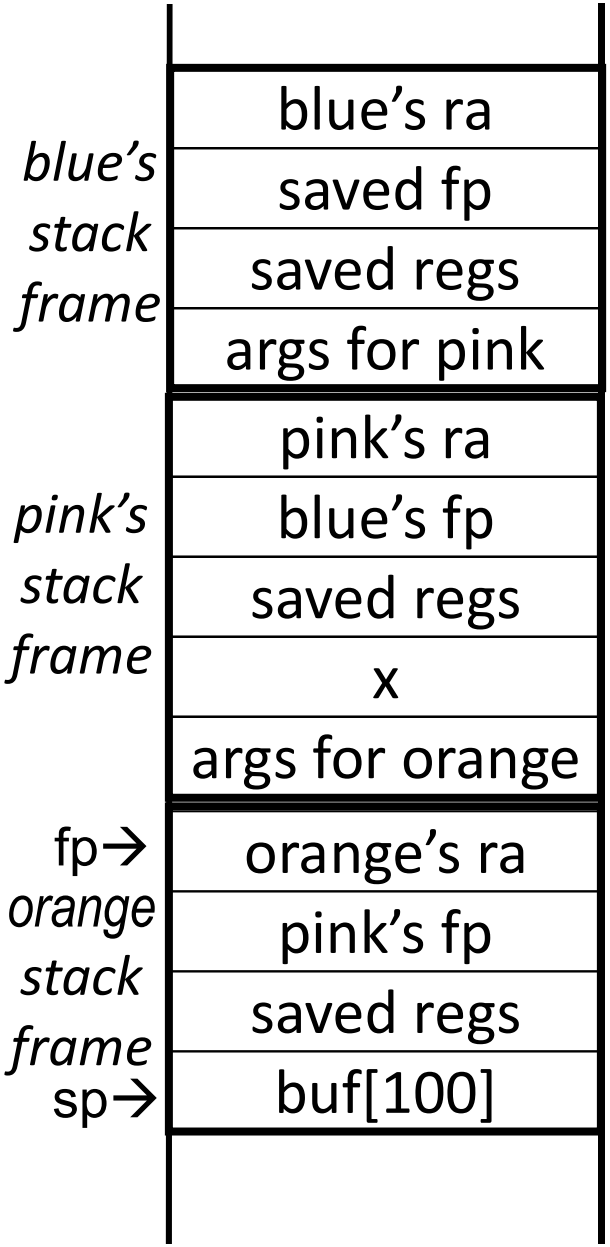
Frame Layout on Stack




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

pink(int a, int b, int c, int d, int e, int f) {
    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?

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?

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)

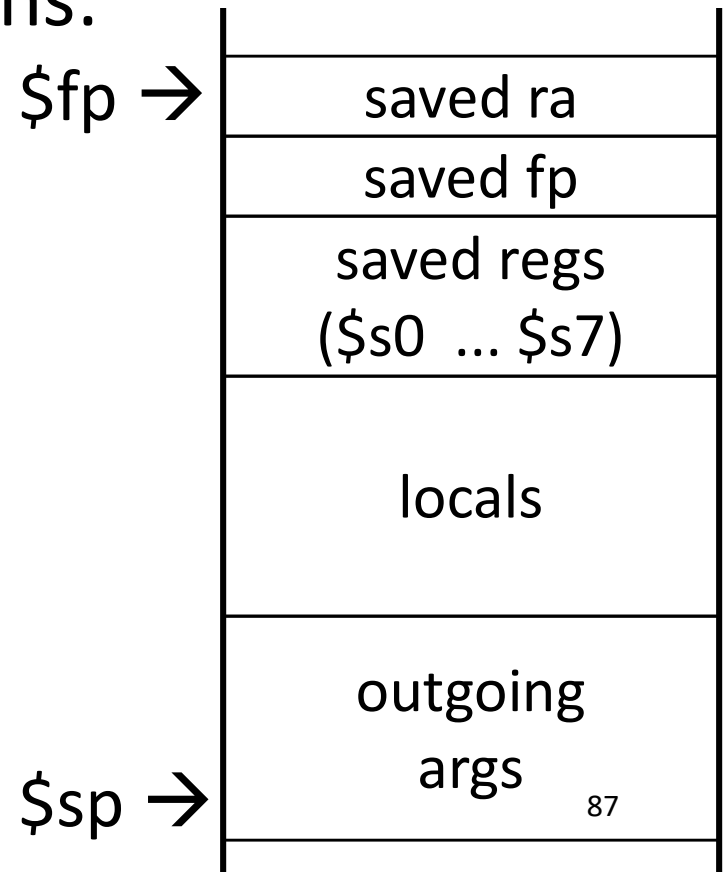
Assembler temporary: $1 (at)

MIPS Register Conventions

r0	$zero	zero
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

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



Activity #1: Calling Convention Examp

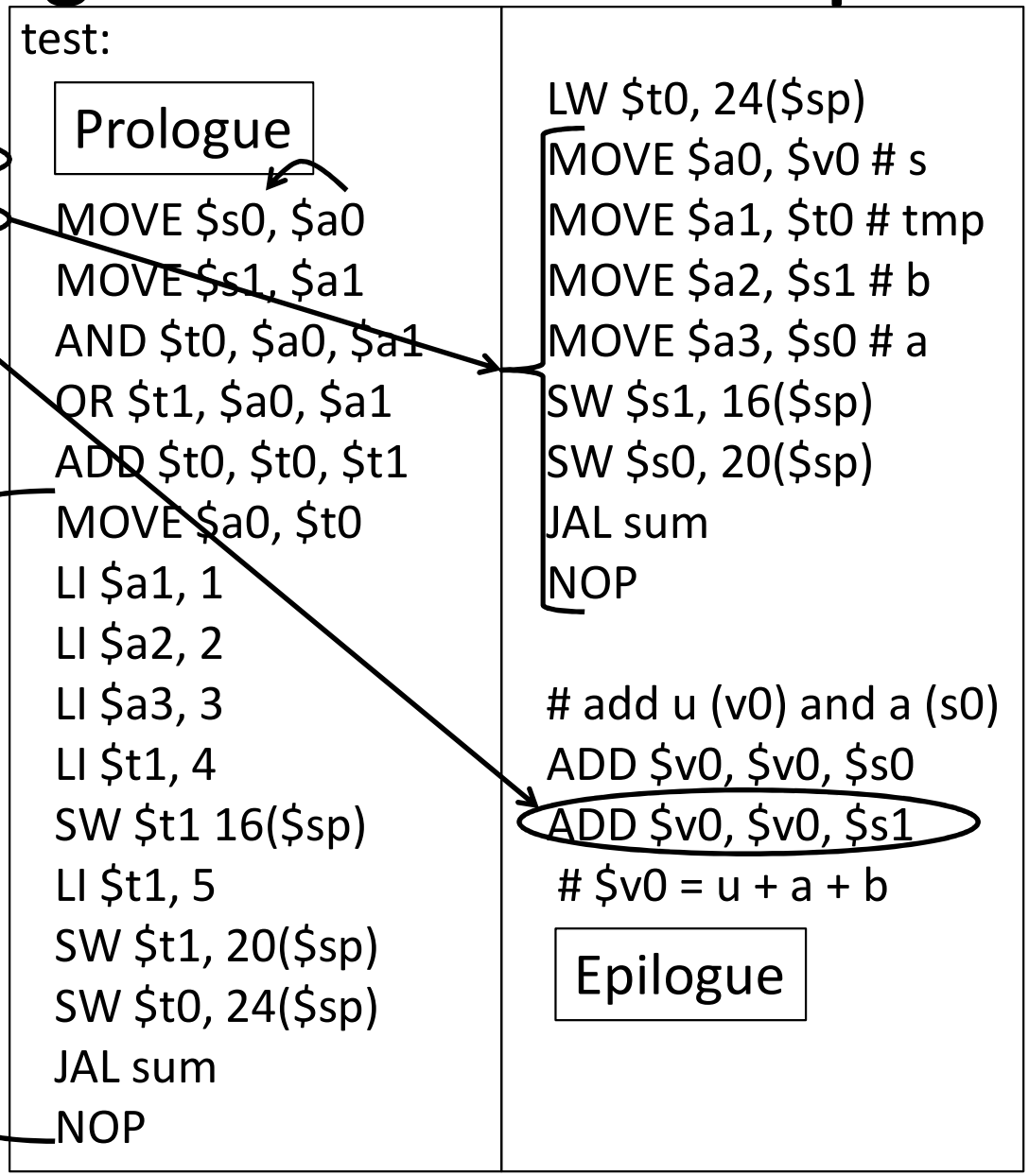
```
int test(int a, int b) {
    int tmp = (a&b)+(a|b);
    int s = sum(tmp,1,2,3,4,5);
    int u = sum(s,tmp,b,a,b,a);
    return u + a + b;
}
```

Correct Order:

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

Activity #1: 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;
}
```

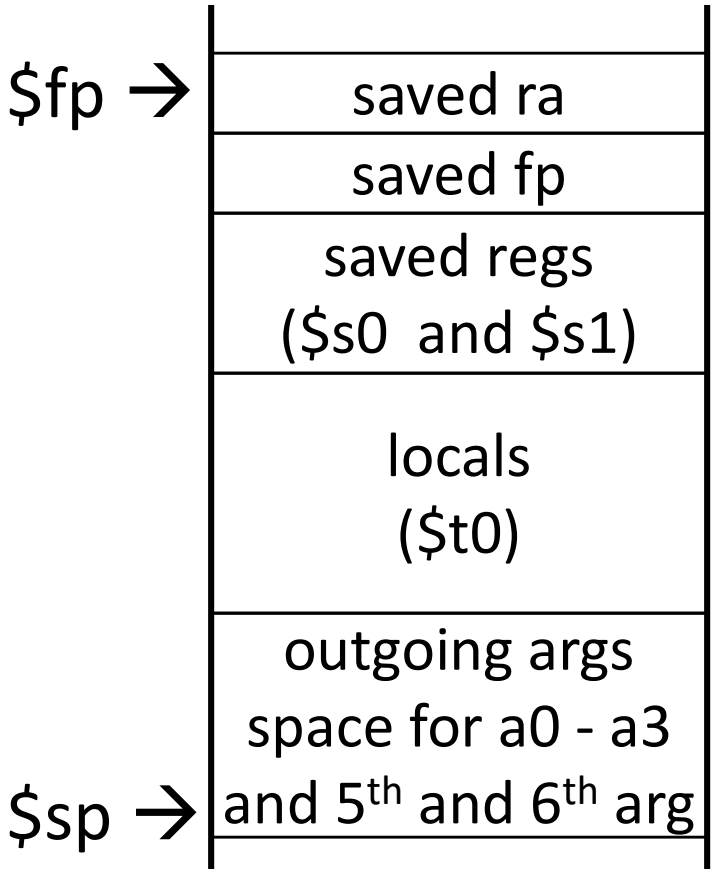
How many bytes do we need to allocate for the stack frame?

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

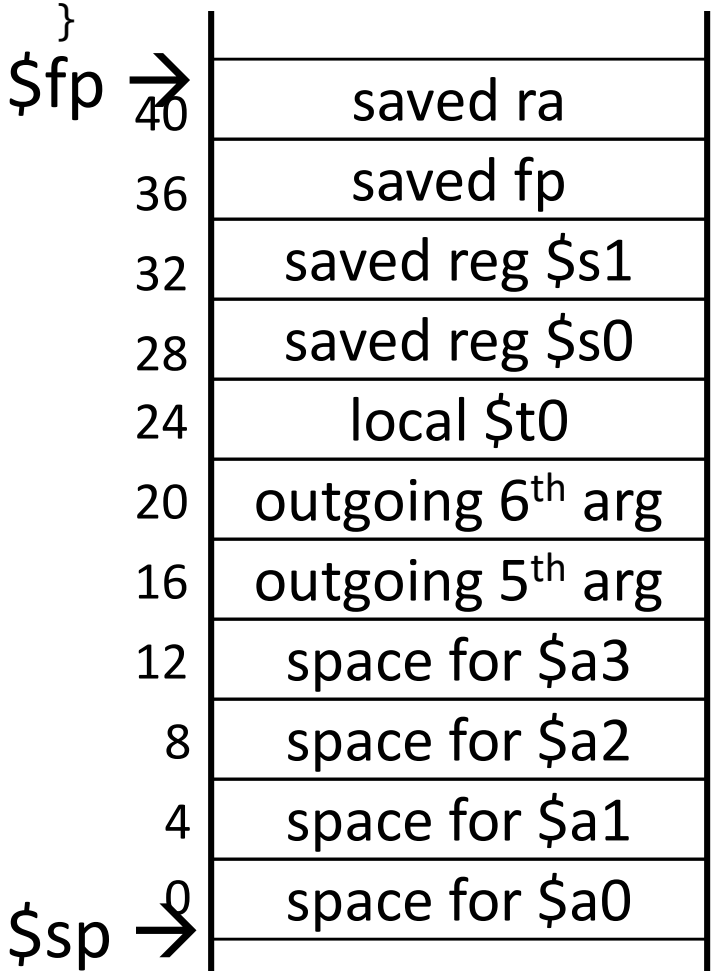
```
MOVE $s0, $a0
MOVE $s1, $a1
AND $t0, $a0, $a1
OR $t1, $a0, $a1
ADD $t0, $t0, $t1
MOVE $a0, $t0
LI $a1, 1
LI $a2, 2
LI $a3, 3
LI $t1, 4
SW $t1 16($sp)
LI $t1, 5
SW $t1, 20($sp)
SW $t0, 24($sp)
JAL sum
NOP
```

```
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:

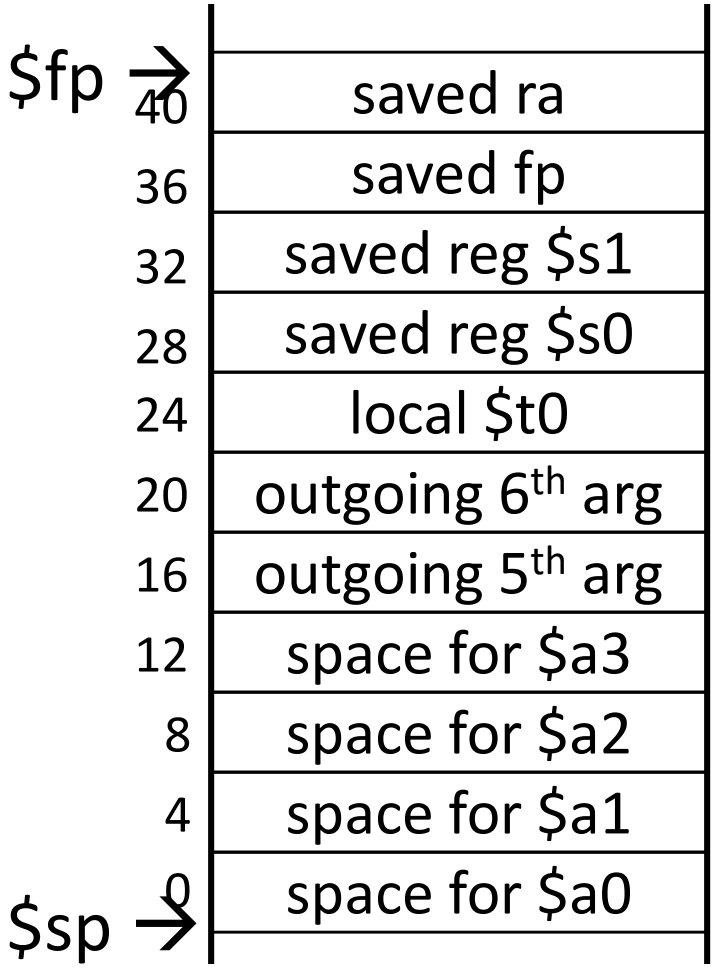
Prologue

```
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

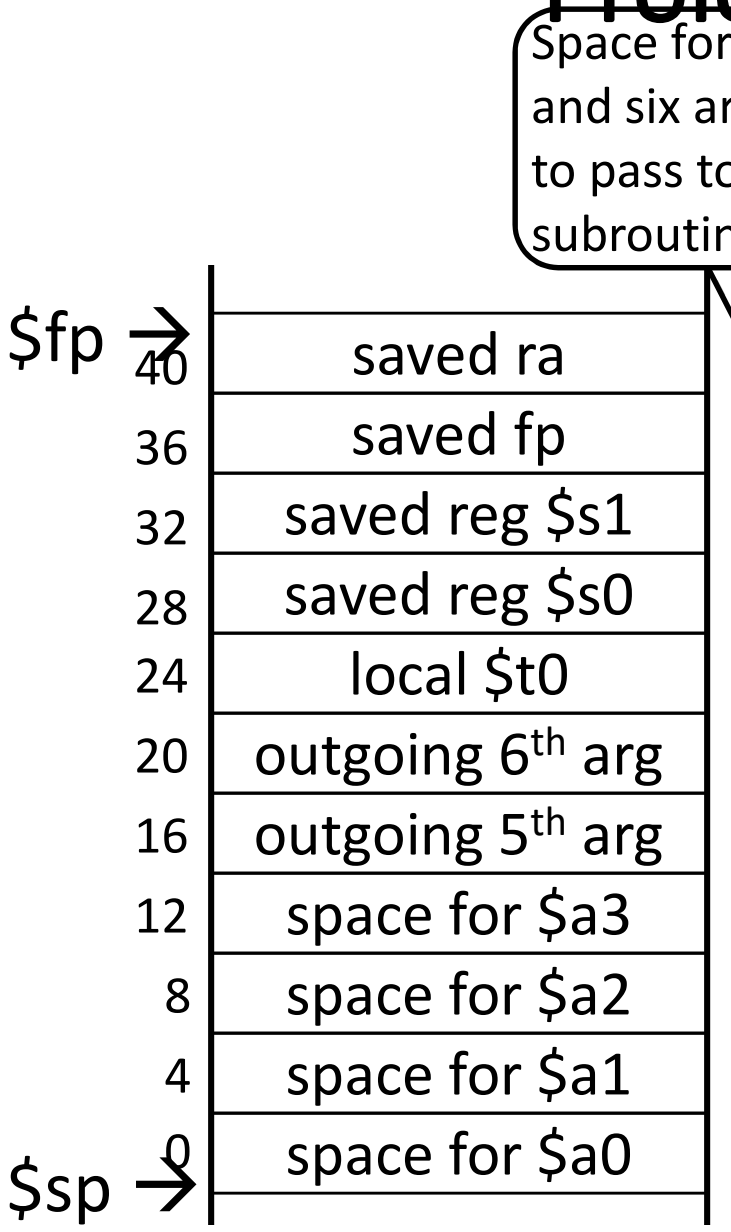
Activity #2: Calling Convention Example: Prologue, Epilogue



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




```
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
```

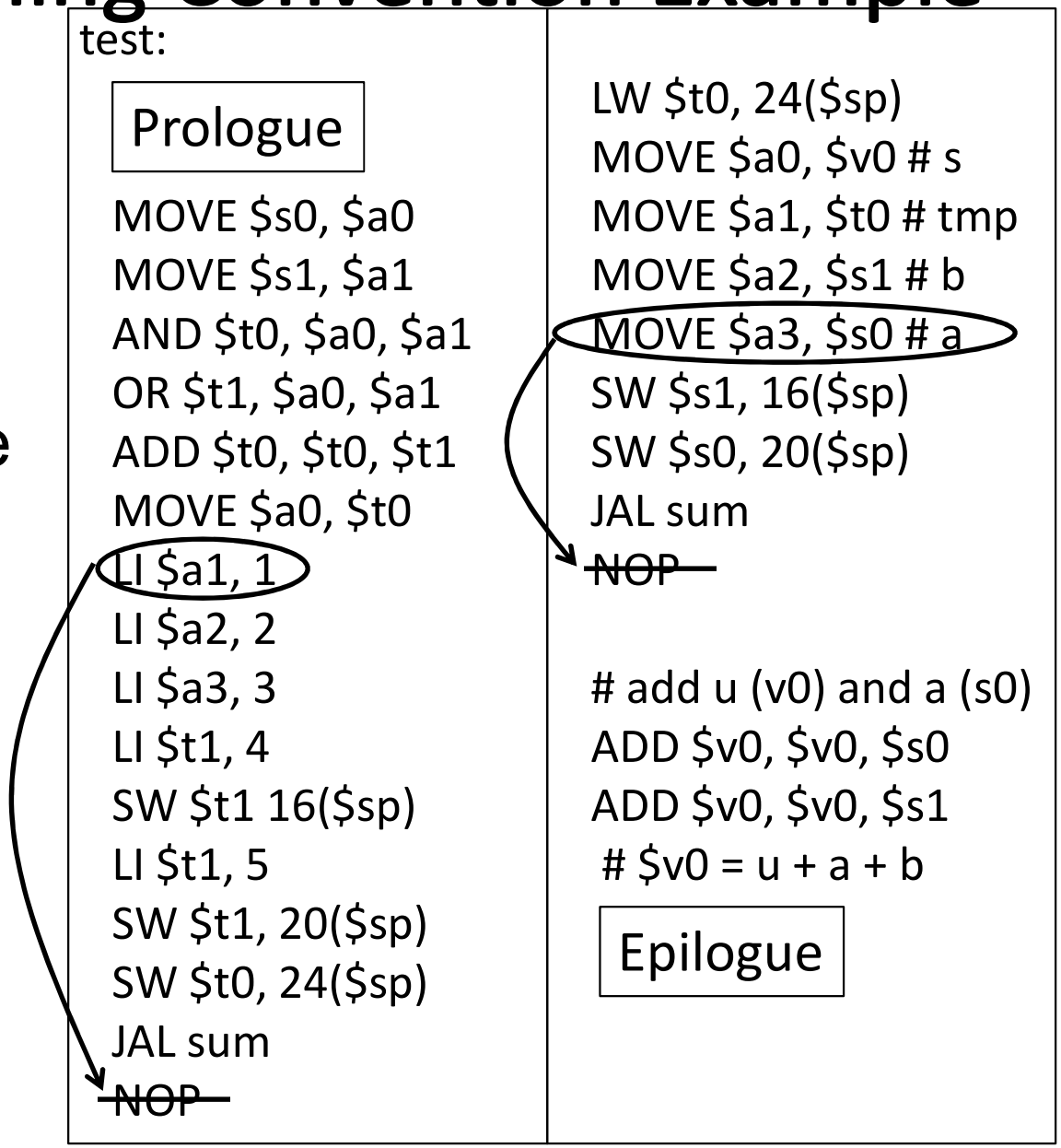
Next Goal

Can we optimize the assembly code at all?

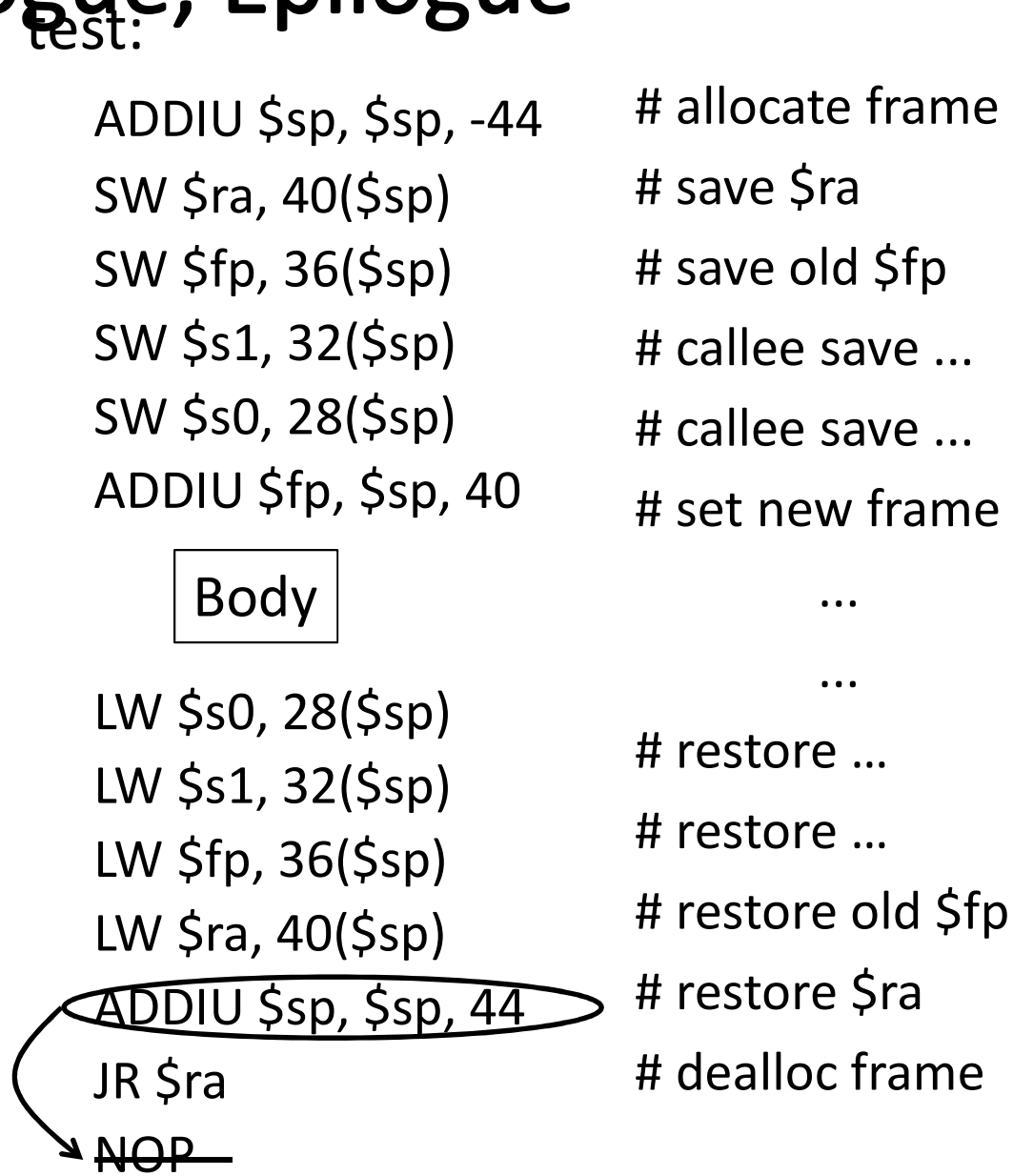
Activity #3: Calling Convention Example

```
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?



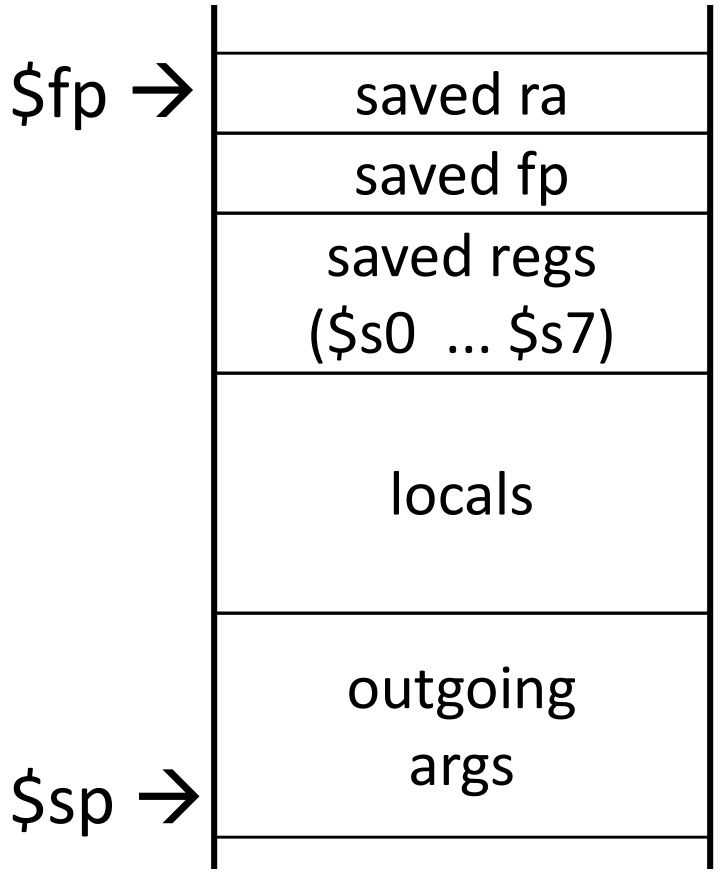
Activity #3: Calling Convention Example: Prologue, Epilogue



Minimum stack size for a standard function?

Minimum stack size for a standard function?

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



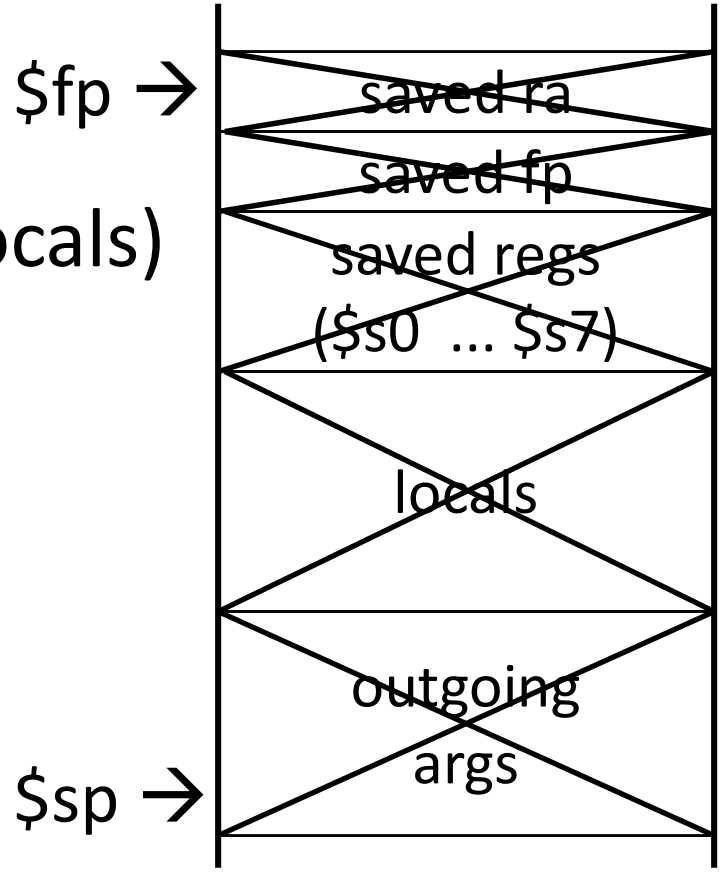
Leaf Functions

Leaf function does not invoke any other functions

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

Optimizations?

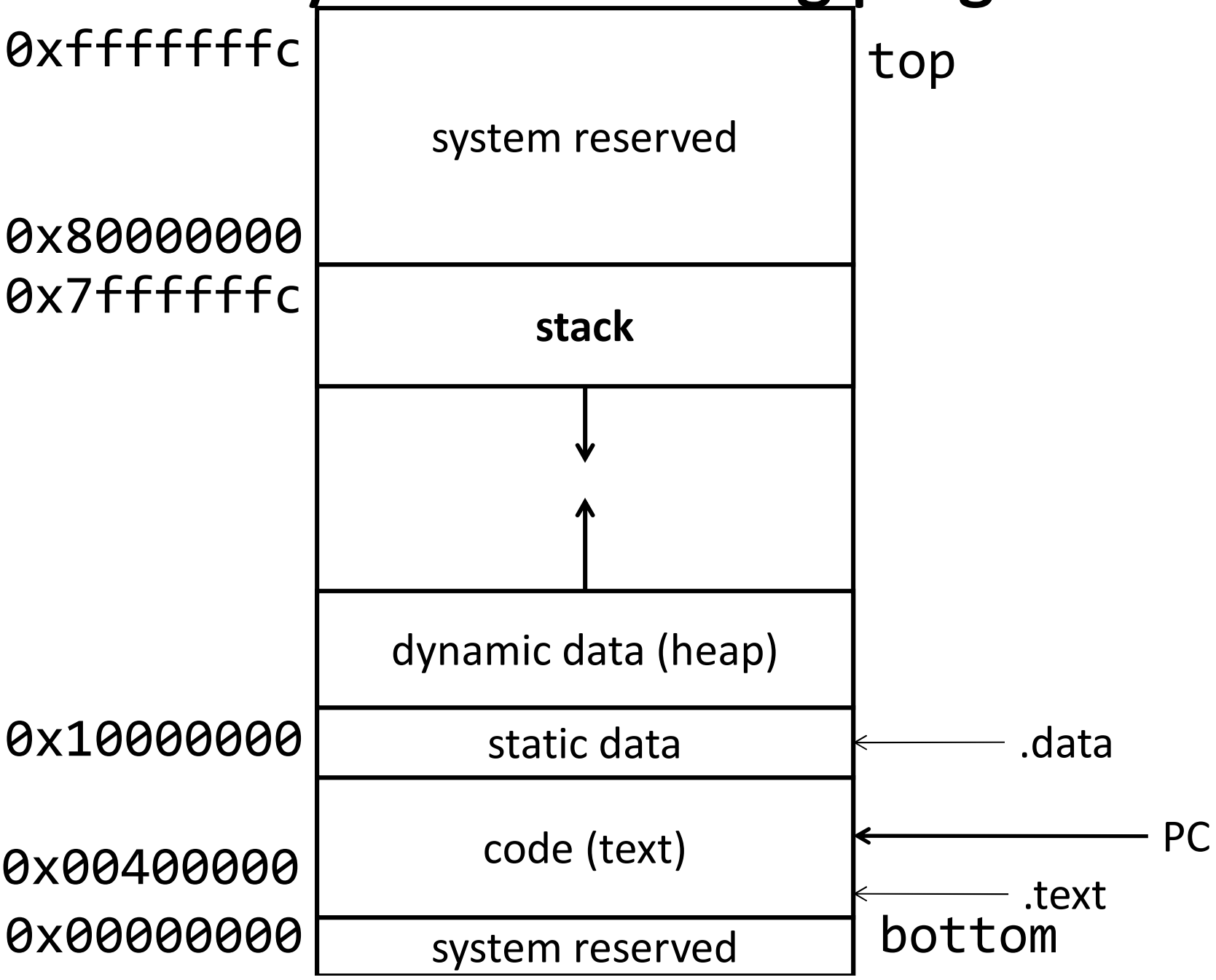
- No saved regs (or locals)
- No outgoing args
- Don't push $ra
- No frame at all?
  - Maybe.



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)?

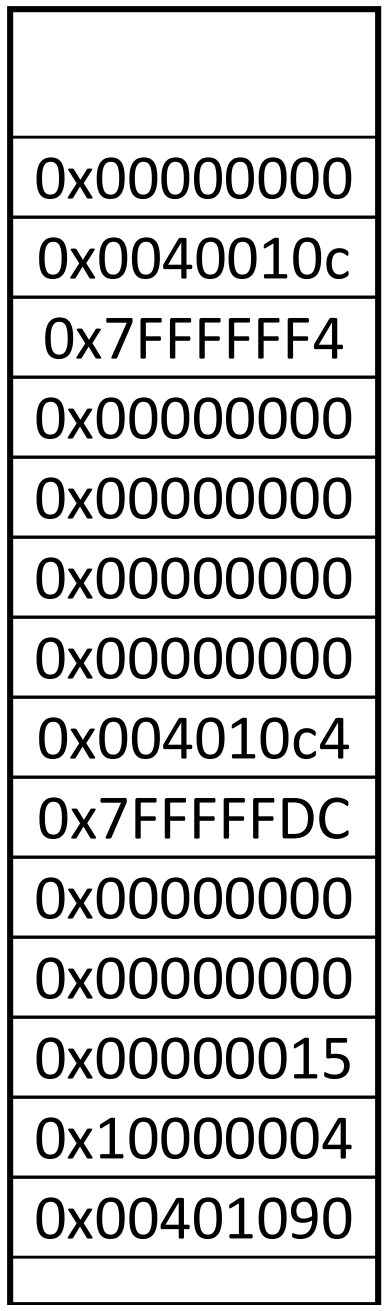
Anatomy of an executing program



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?

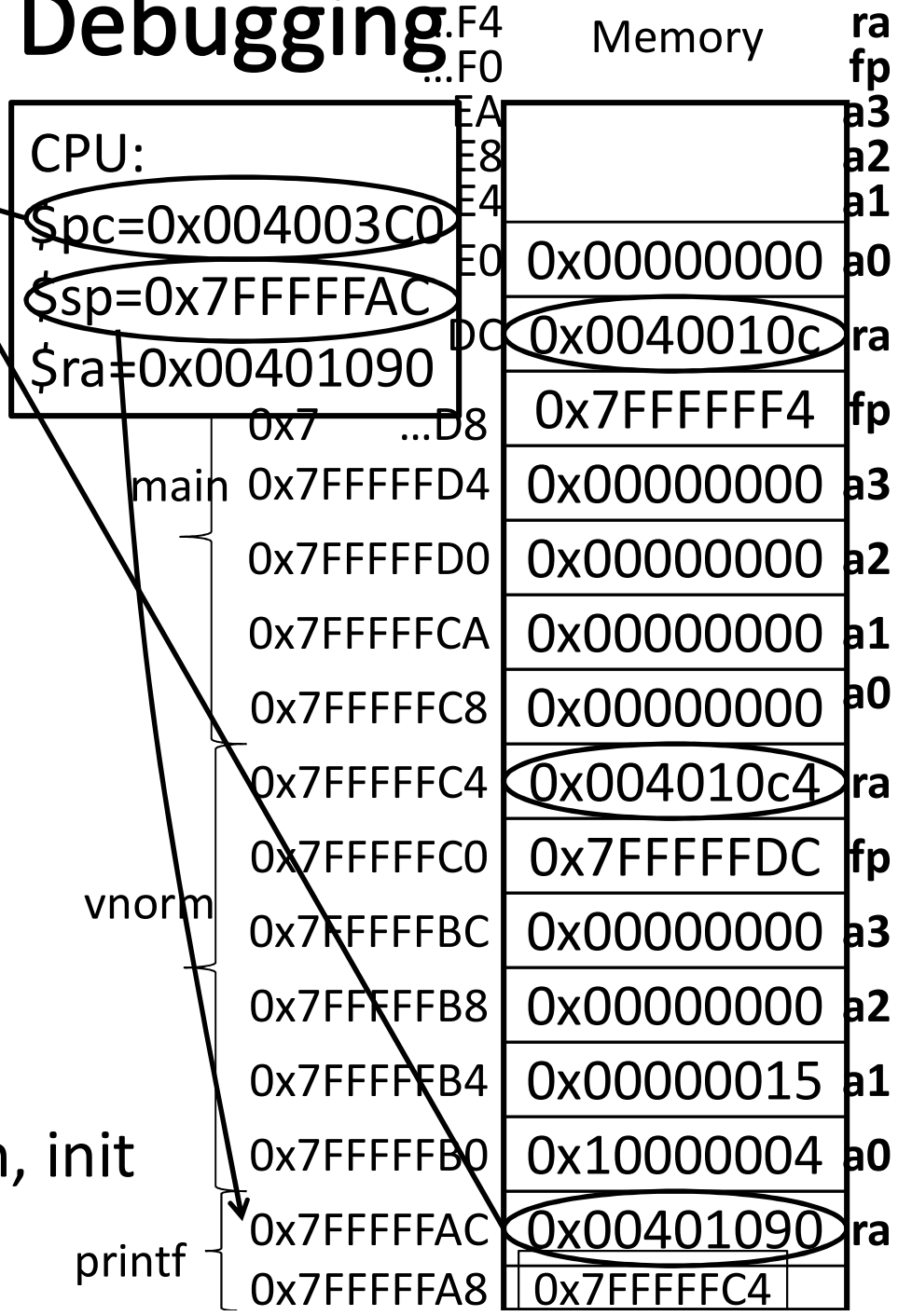
Args?

Stack depth?

Call trace?

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, $v1
- stack frame ($fp to $sp) contains:
  - $ra (clobbered on JAL to sub-functions)
  - $fp
  - 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