Calling Conventions

Prof. Kavita Bala and Prof. Hakim Weatherspoon

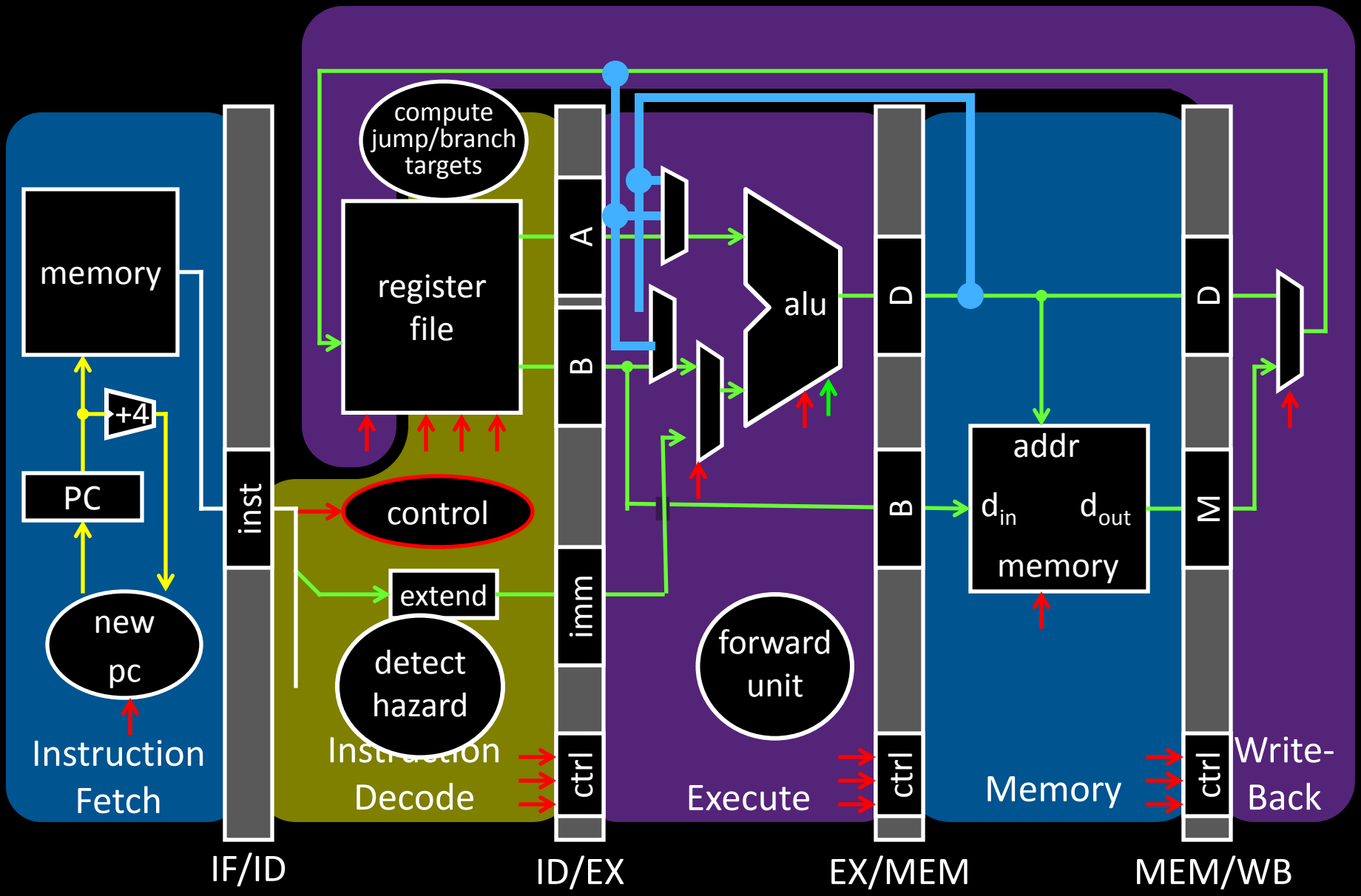
CS 3410, Spring 2014

Computer Science

Cornell University

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

Big Picture: Where are we now?



Big Picture: Where are we going?

C

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

compiler

MIPS assembly

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

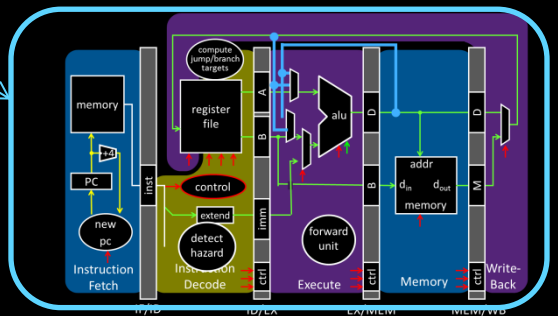
r0 = 0
r5 = r0 + 10
r5 = r5<<1 #r5 = r5 * 2
r5 = r15 + 15

assembler

machine code

```
op = addi   r0    r5                    10
0010000000000101000000000001010
0000000000000101001010000100000
0010000010100101000000000001111
op = addi   r5    r5                    15
op = r-type         r5    r5   shamt=1   func=sll
```

CPU



Circuits

Gates

Transistors

Silicon

Goals for Today

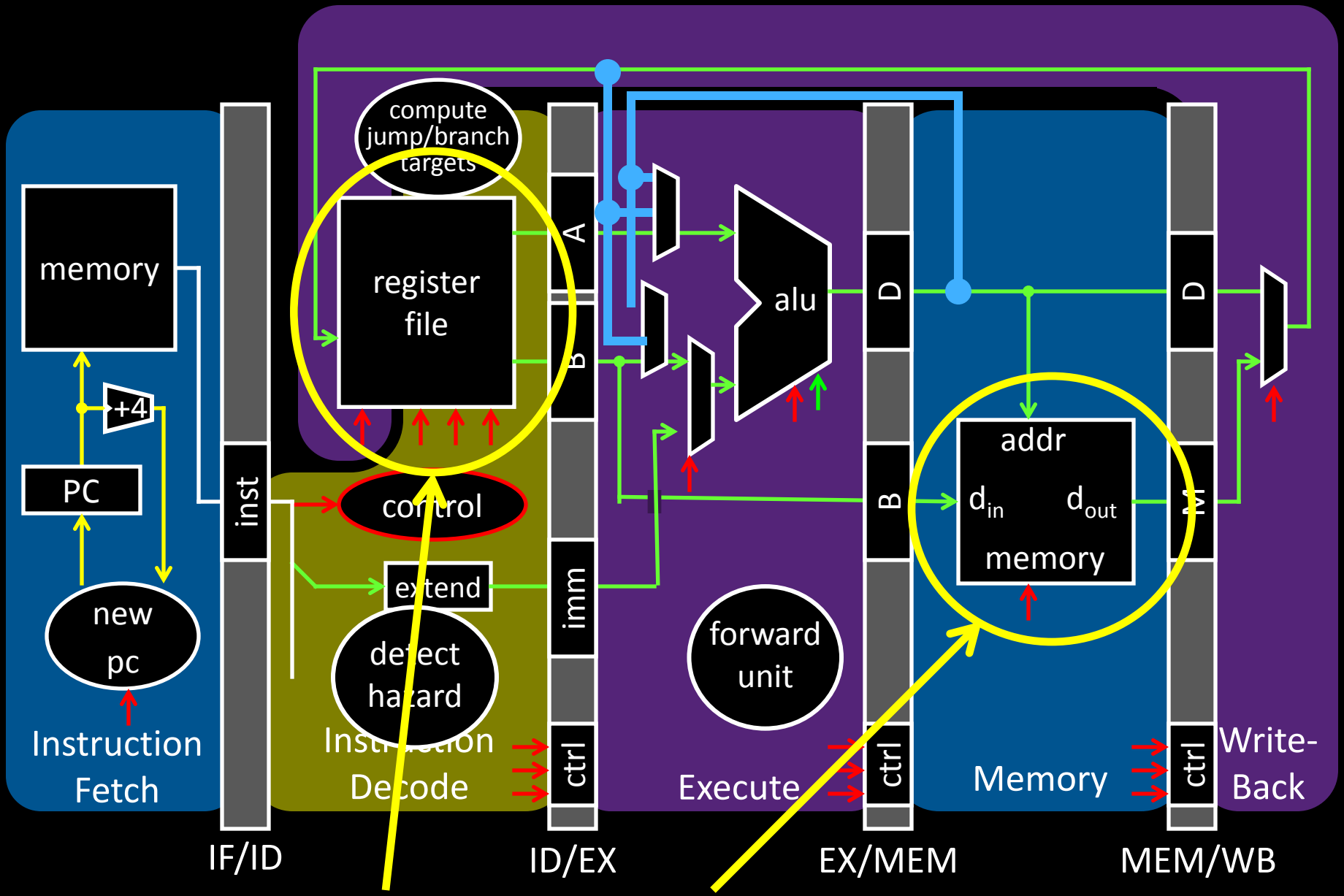
Calling Convention for Procedure Calls

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

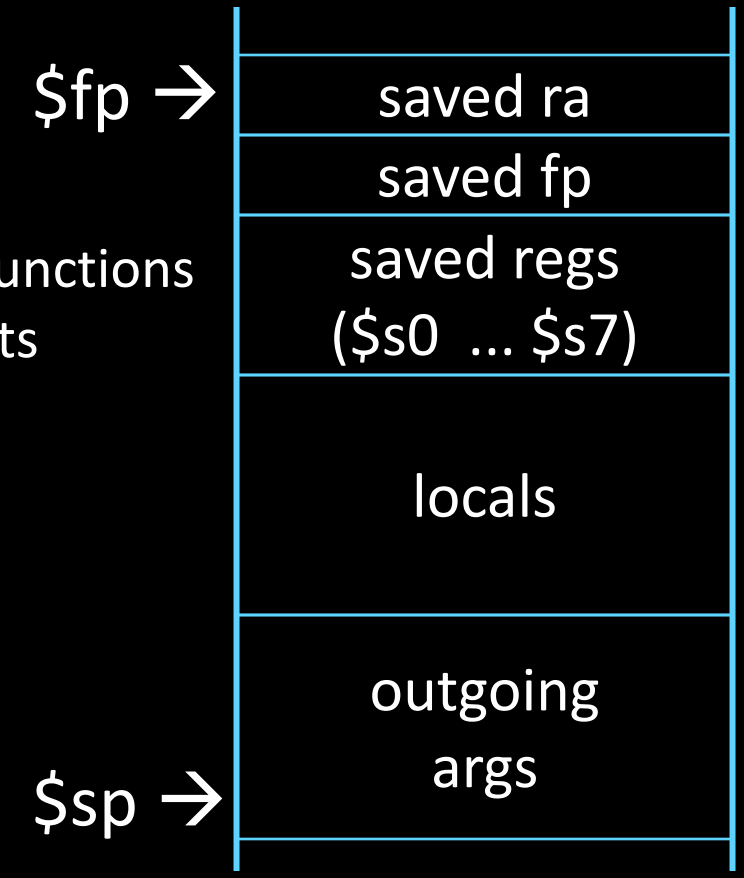
Cheat Sheet and Mental Model for Today



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

Goals for Today

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

What is the convention to call a subroutine?

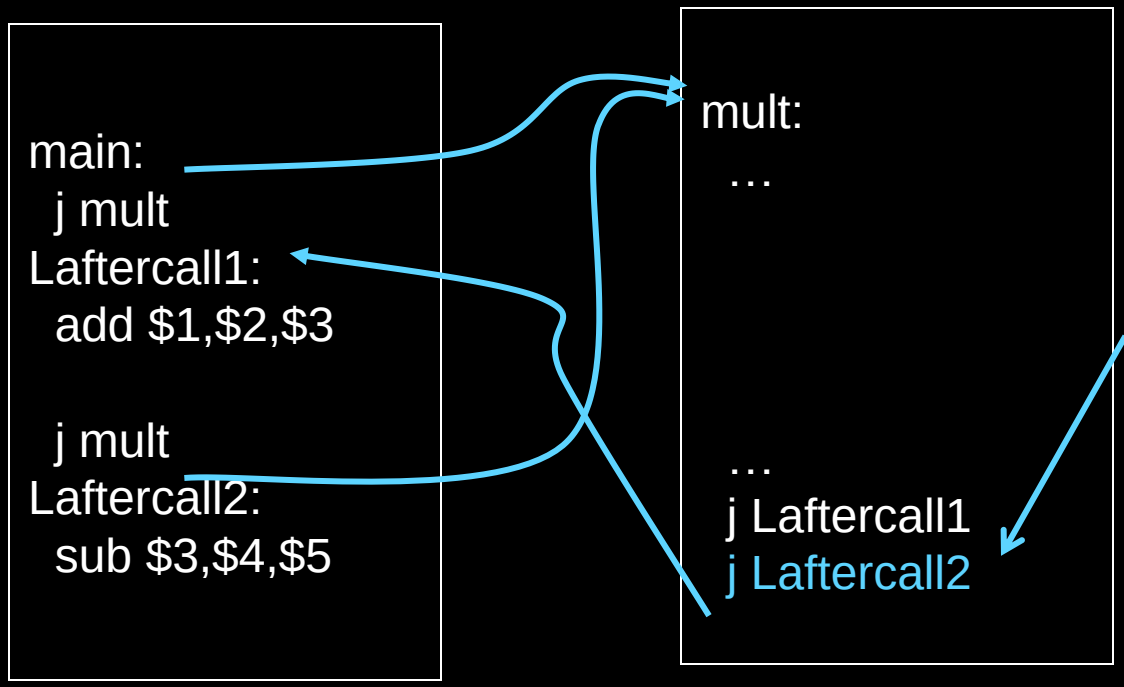
```
int main (int argc, char* argv[ ]) {
      int n = 9;
      int result = multi(n);
}

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

Procedure Call – Attempt #1: Use Jumps

Jumps and branches can transfer control to the callee (called procedure)

Jumps and branches can transfer control back

Procedure Call – Attempt #1: Use Jumps



Jumps and branches can transfer control to the callee (called procedure)

Jumps and branches can transfer control back

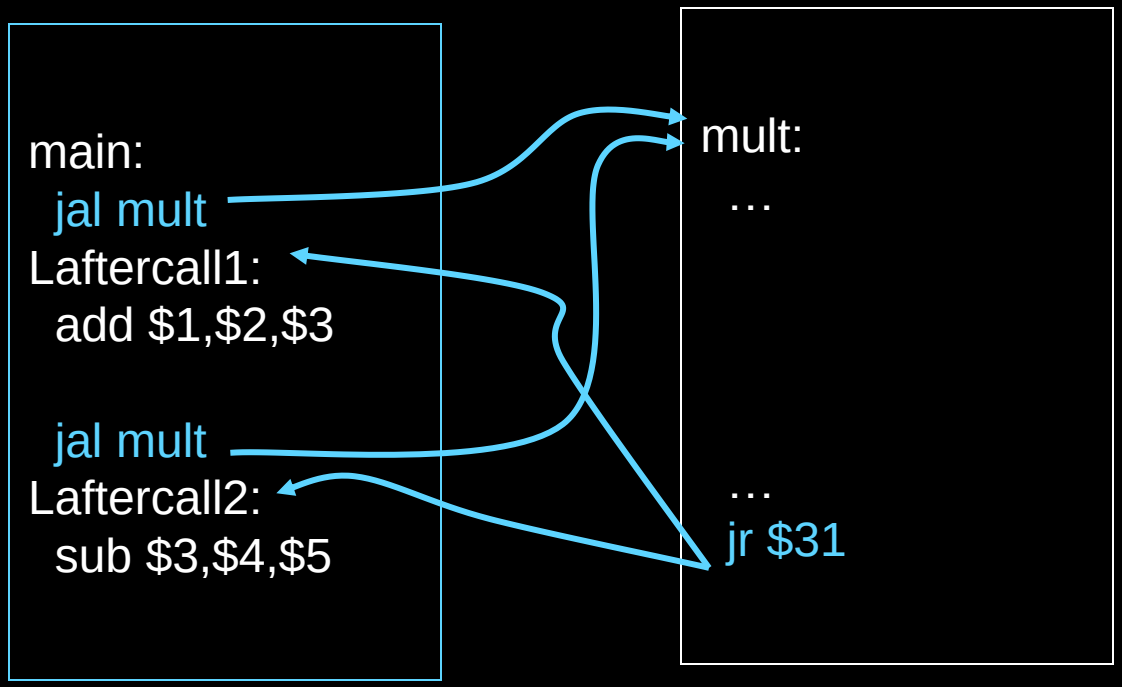
What happens when there are multiple calls from different call sites?

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

Procedure Call – Attempt #2: JAL/JR

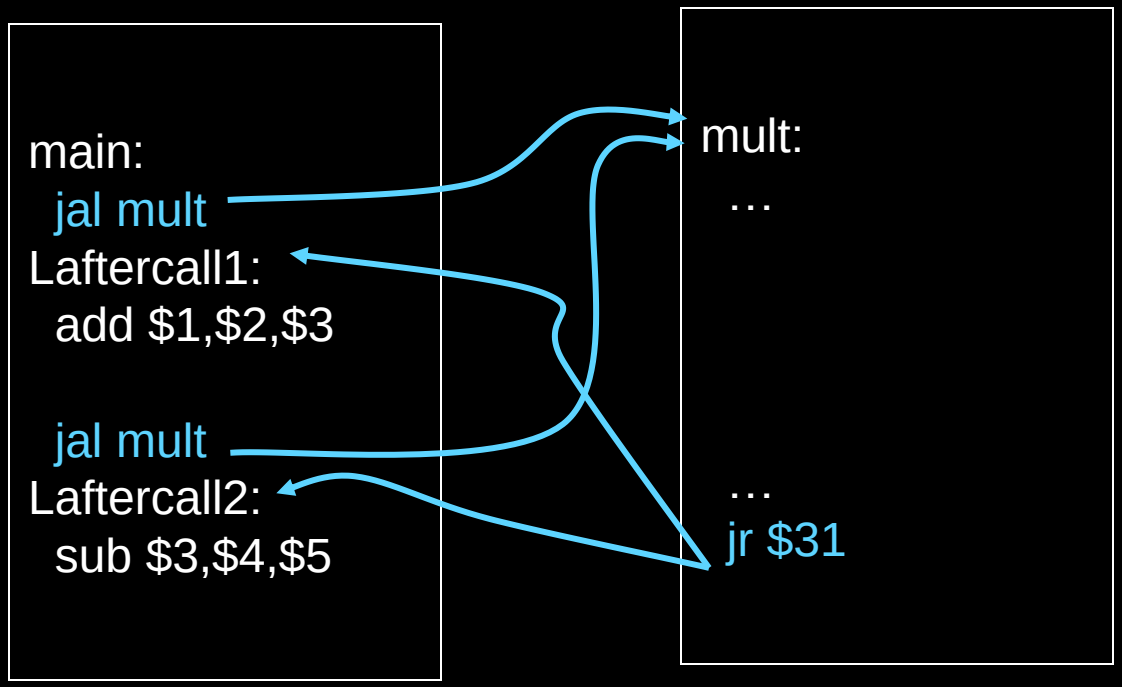


JAL saves the PC in register $31

Subroutine returns by jumping to $31

What happens for recursive invocations?

Procedure Call – Attempt #2: JAL/JR



JAL saves the PC in register $31

Subroutine returns by jumping to $31

What happens for recursive invocations?

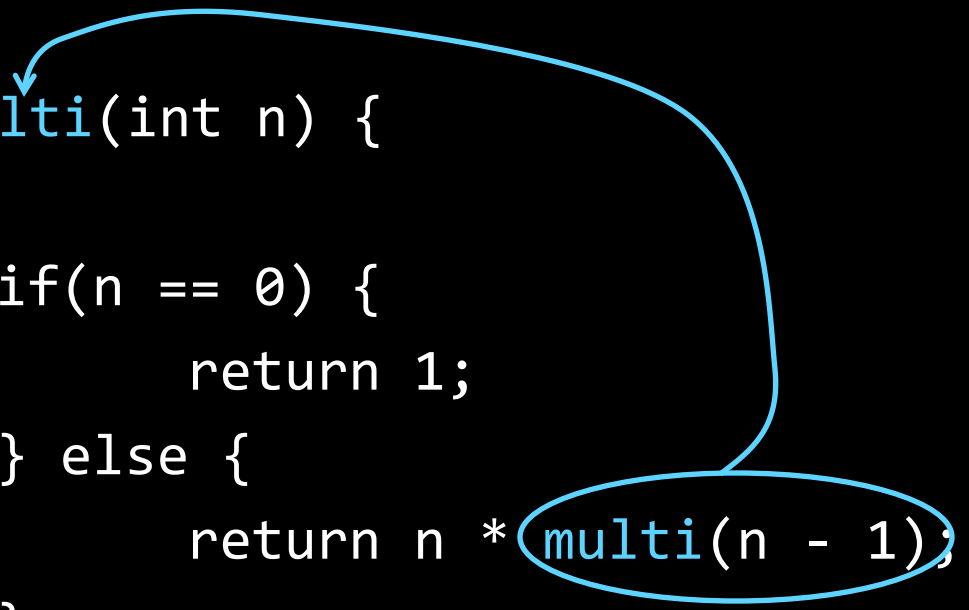
Procedure Call – Attempt #2: JAL/JR

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


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

Procedure Call – Attempt #2: JAL/JR

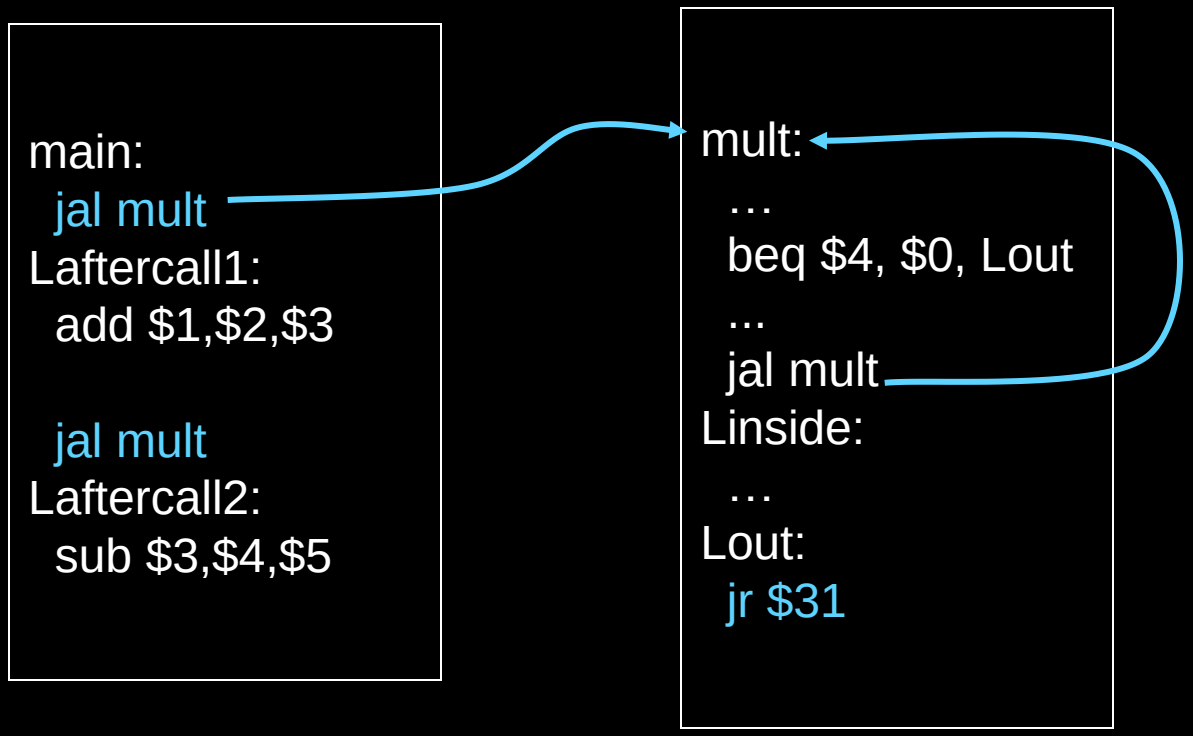
```
int main (int argc, char* argv[ ]) {
      int n = 9;
      int result = multi(n);
}
```

What happens for recursive invocations?



```
int multi(int n) {

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

Procedure Call – Attempt #2: JAL/JR



What happens for recursive invocations?

Recursion overwrites contents of $31

Need to save and restore the register contents

Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

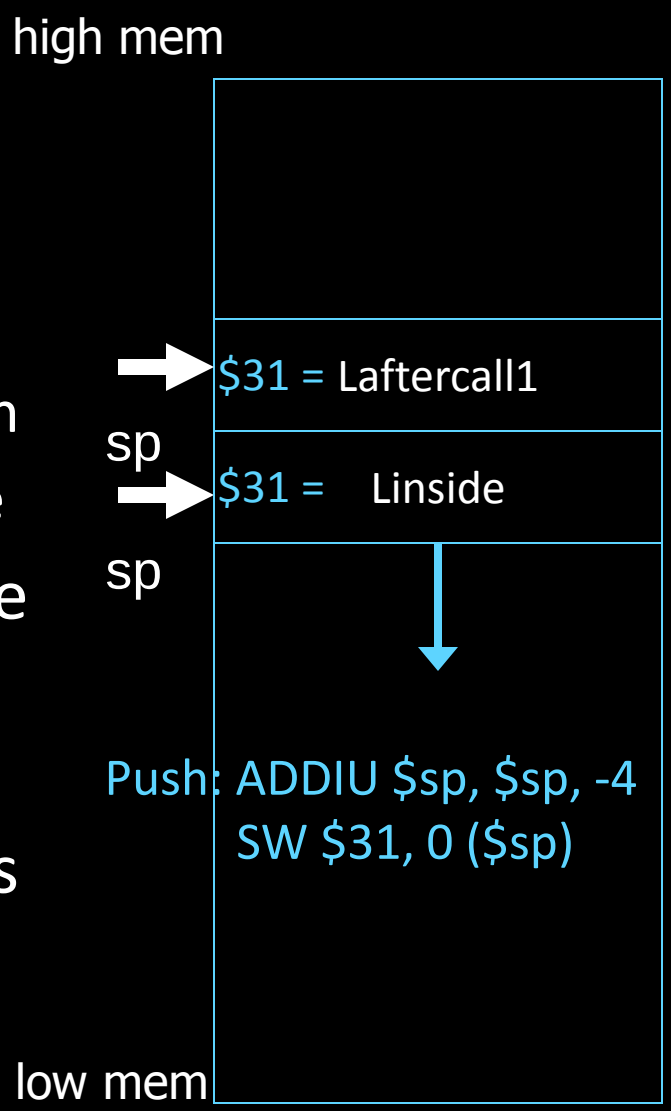
- the return address for that invocation
- the local variables for that procedure

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



Need a "Call Stack"

Call stack

- contains activation records (aka stack frames)

Each activation record contains

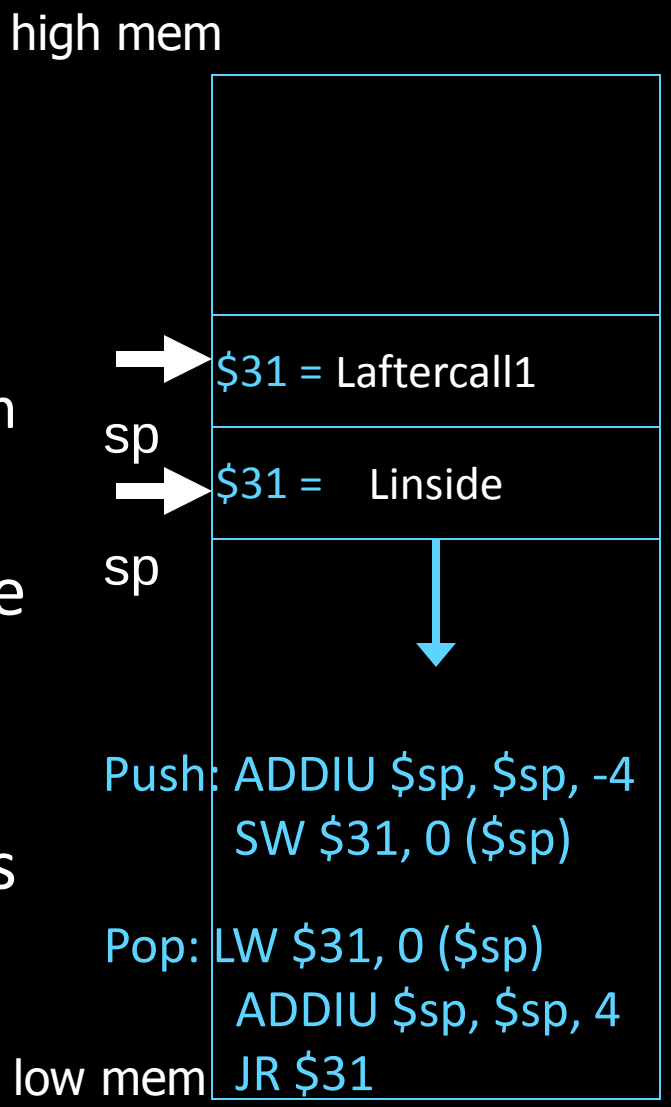
- the return address for that invocation
- the local variables for that procedure

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



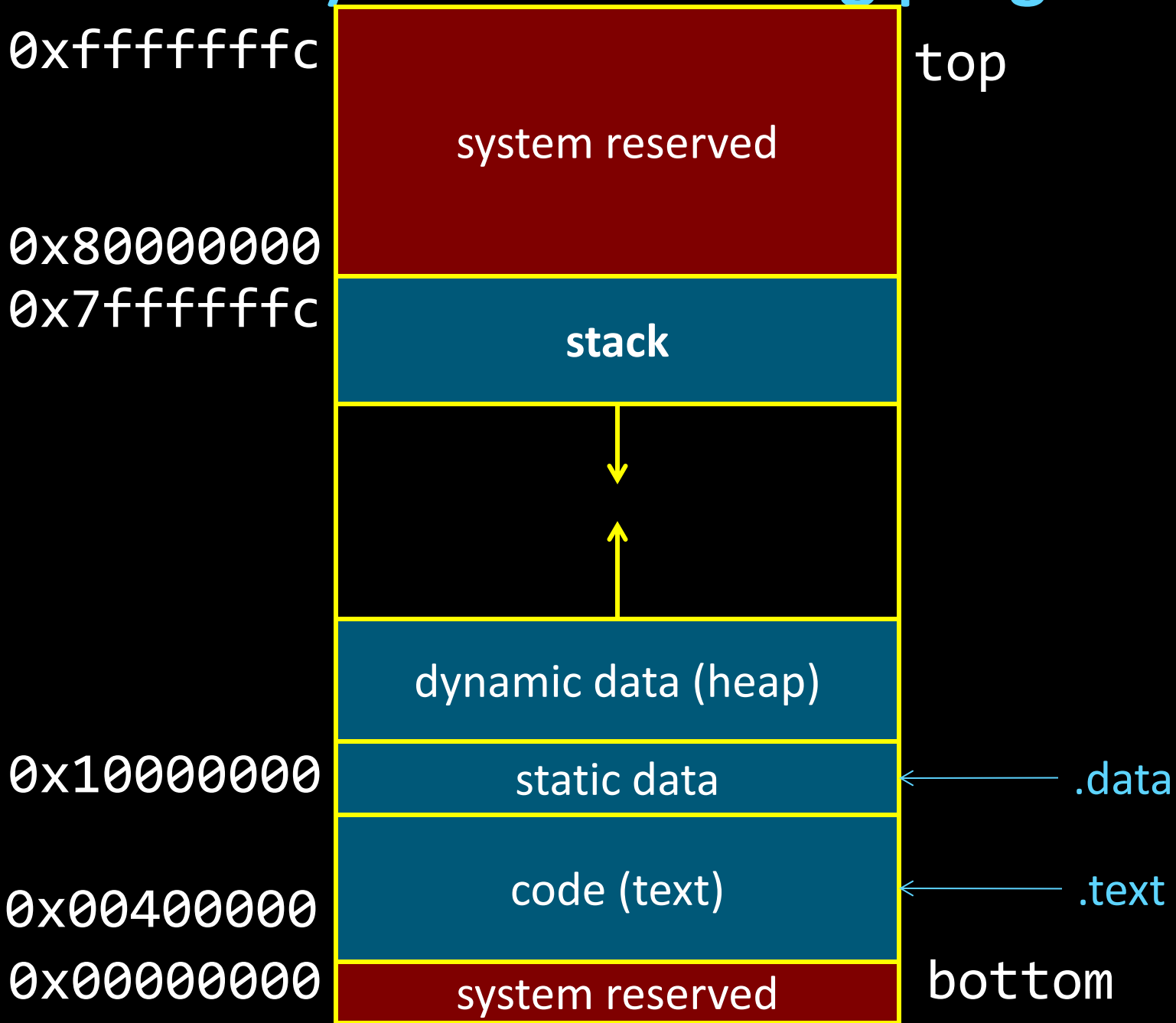
Stack Growth

(Call) Stacks start at a high address in memory

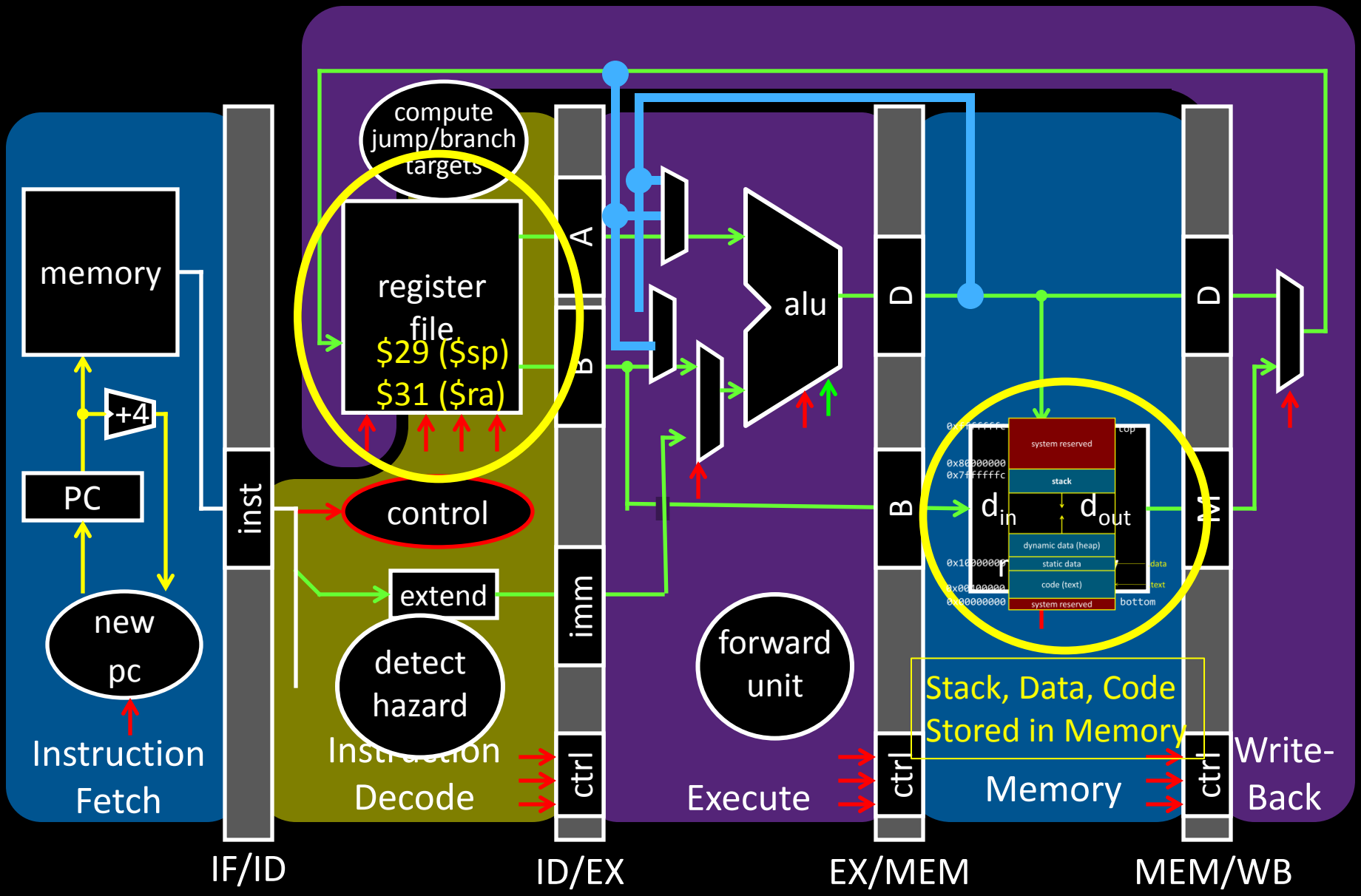
Stacks grow down as frames are pushed on

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

Anatomy of an executing program



Anatomy of an executing program

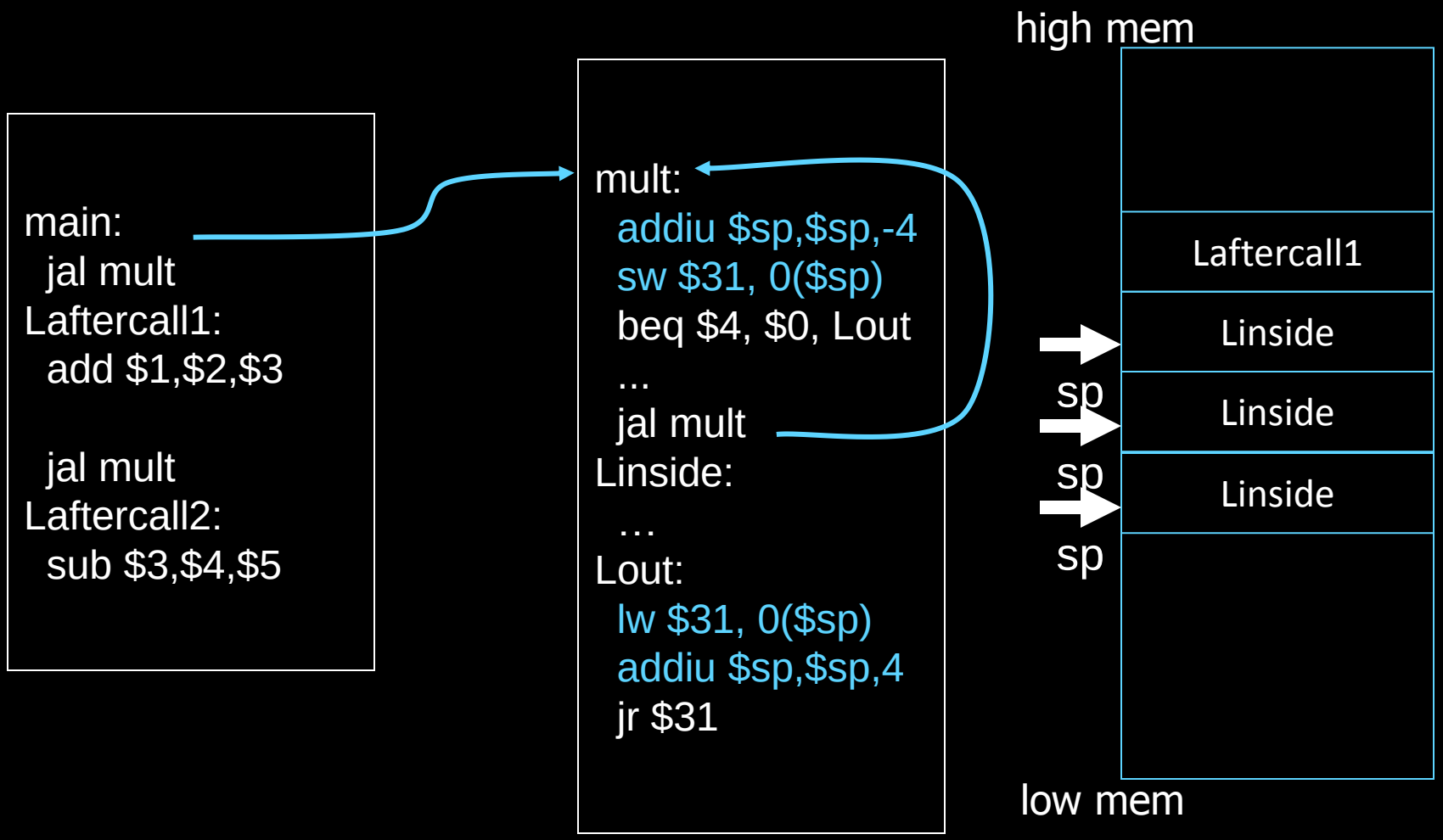


Takeaway2: Need a Call Stack

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

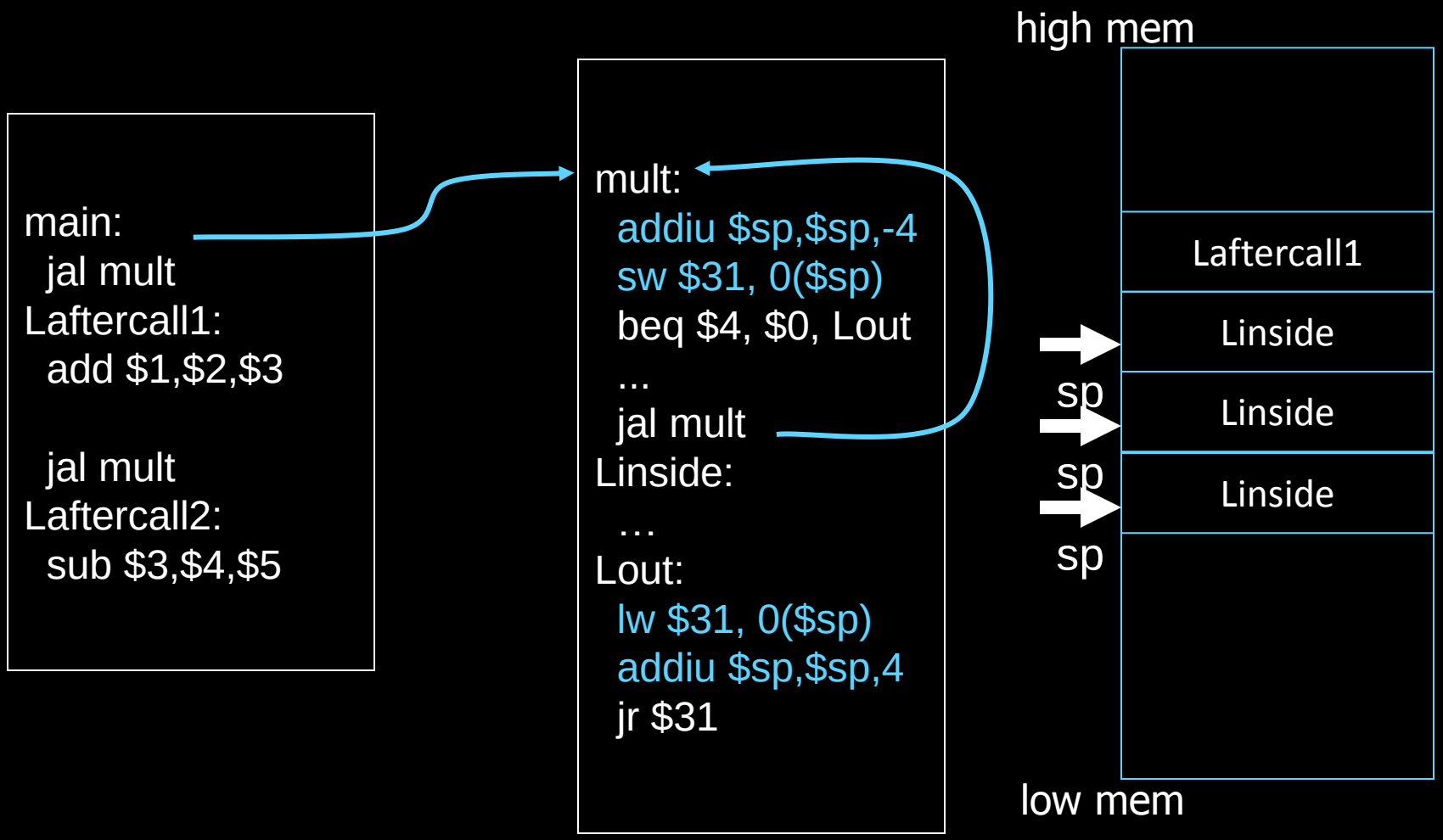
Need a Call Stack to return to correct calling procedure. To maintain a stack, need to store an ***activation record*** (aka a "stack frame") in memory. Stacks keep track of the correct return address by storing the contents of $31 in memory (the stack).

Attempt #3: JAL/JR with Activation Records



Stack used to save and restore contents of $31

Attempt #3: JAL/JR with Activation Records



Stack used to save and restore contents of $31

How about arguments?

Next Goal

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

Attempt #4: 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);`
- `int subf(int a, int b, int c, int d, int e);`
- `int isalpha(char c);`
- `int treesort(struct Tree *root);`
- `struct Node *createNode();`
- `struct Node mynode();`

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

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

Simple Argument Passing

```
main:
   li $a0, 6
   li $a1, 7
   jal min
   // result in $v0
```

First four arguments are passed in registers

- Specifically, $4, $5, $6 and $7, aka $a0, $a1, $a2, $a3

The returned result is passed back in a register

- Specifically, $2, aka $v0

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?

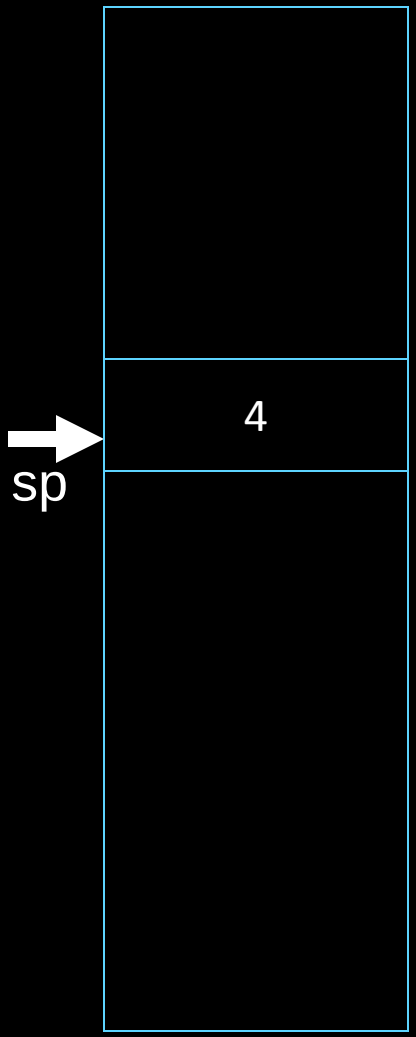
Many Arguments

```
main:
   li $a0, 0
   li $a1, 1
   li $a2, 2
   li $a3, 3
   jal subf
   // result in $v0
```

What if there are more than 4 arguments?

Many Arguments

```
main:
   li $a0, 0
   li $a1, 1
   li $a2, 2
   li $a3, 3
   li $8, 4
   addiu $sp,$sp,-4
   sw $8, 0($sp)
   jal subf
   // result in $v0
```



What if there are more than 4 arguments?

Use the stack for the additional arguments

- “spill”

Many Arguments

```
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 subf
   // result in $v0
```

What if there are more than 4 arguments?

Use the stack for the additional arguments

- "spill"

Variable Length Arguments

```
printf("Coordinates are: %d %d %d\n", 1, 2, 3);
```

Could just use the regular calling convention, placing first four arguments in registers, spilling the rest onto the stack

- Callee requires special-case code
- if(argno == 1) use a0, … else if (argno == 4) use a3, else use stack offset

Best to use an (initially confusing but ultimately simpler) approach:

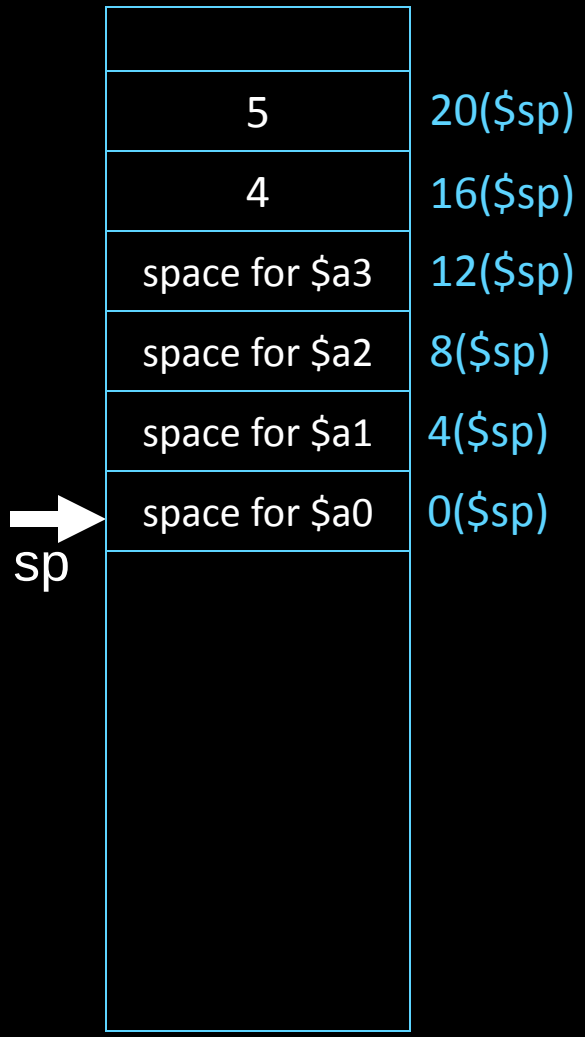
- Pass the first four arguments in registers, as usual
- Pass the rest on the stack
- Reserve space on the stack for all arguments, including the first four

Simplifies functions that use variable-length arguments

- Store a0-a3 on the slots allocated on the stack, refer to all argument

Register Layout on Stack

```
main:
   li $a0, 0
   li $a1, 1
   li $a2, 2
   li $a3, 3
   addiu $sp,s$p,-24
   li $8, 4
   sw $8, 16($sp)
   li $8, 5
   sw $8, 20($sp)
   jal subf
   // result in$ v0
```

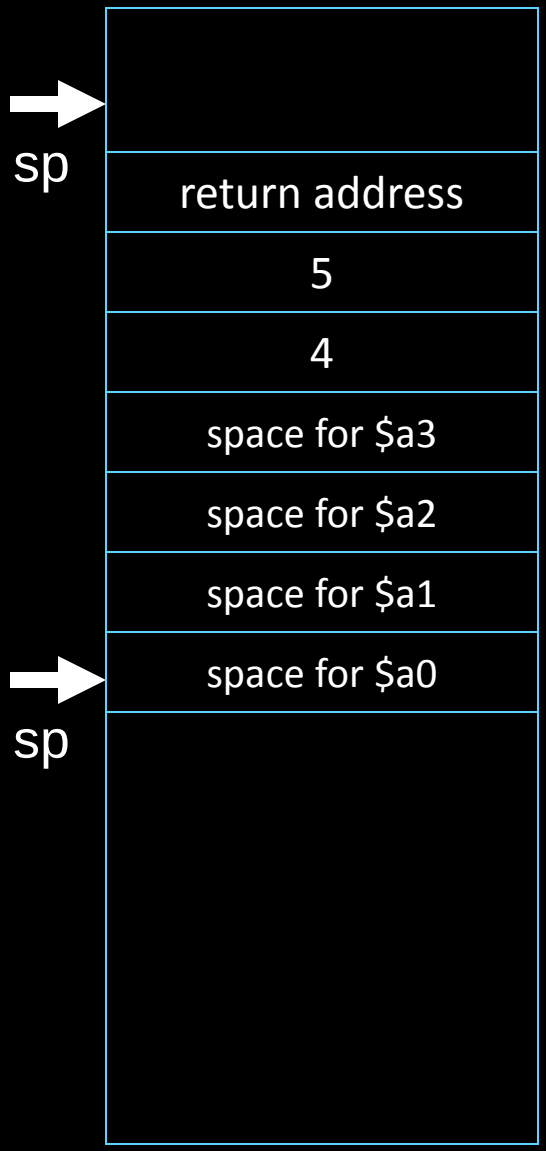


First four arguments are in registers

The rest are on the stack

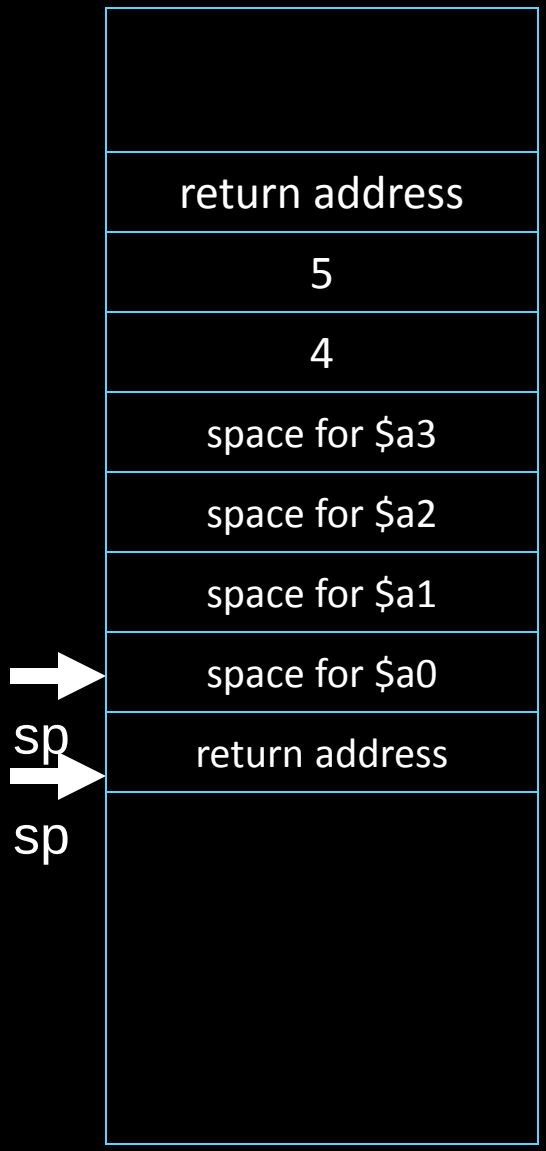
There is room on the stack for the first four arguments, just in case

Frame Layout on Stack



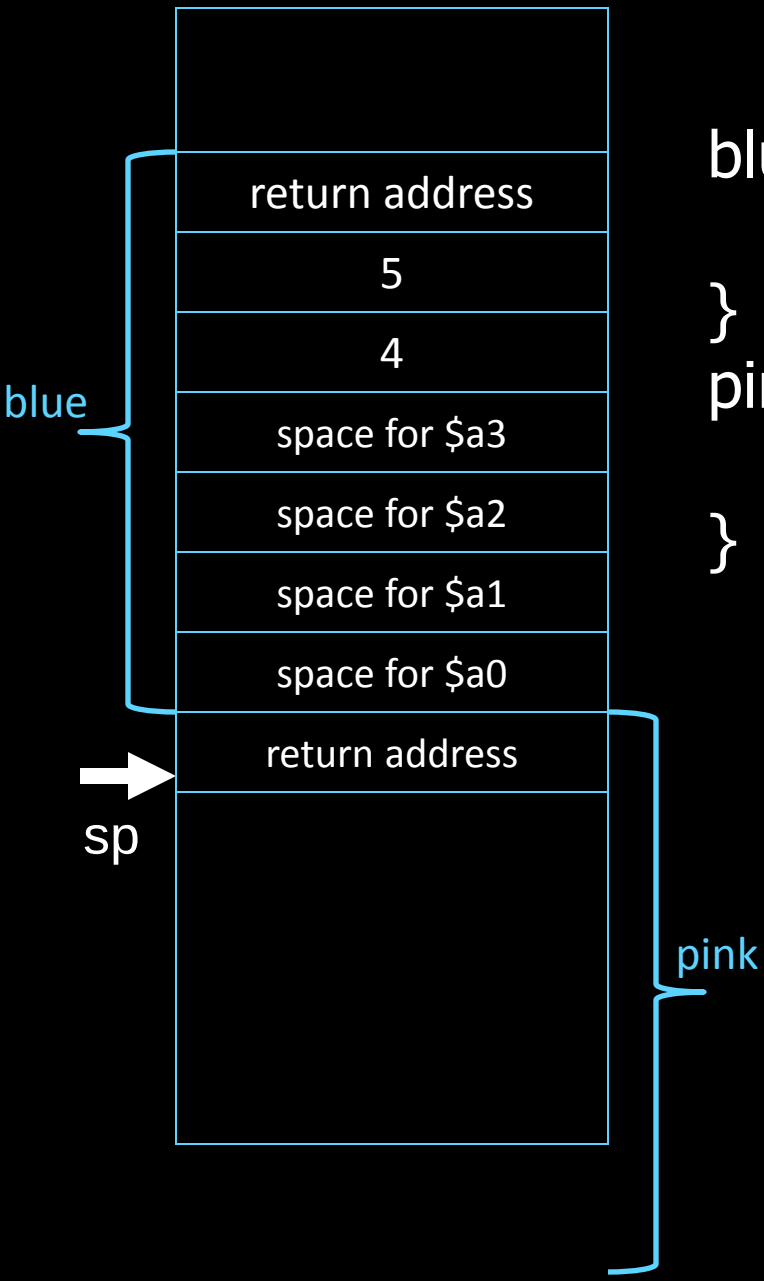

```
blue() {
   pink(0,1,2,3,4,5);
}
```

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) {
     ...
}
```

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) {
     ...
}
```

Conventions so far:

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed on the stack
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains extra arguments to sub-functions
  - contains **space** for first 4 arguments to sub-functions

MIPS Register Conventions so far:

r0	$zero	zero
r1	$at	assembler temp
r2	$v0	function
r3	$v1	return values
r4	$a0	
r5	$a1	function
r6	$a2	arguments
r7	$a3	
r8		
r9		
r10		
r11		
r12		
r13		
r14		
r15		

r16		
r17		
r18		
r19		
r20		
r21		
r22		
r23		
r24		
r25		
r26	$k0	reserved
r27	$k1	for OS kernel
r28		
r29		
r30		
r31	$ra	return address

Pseudo-Instructions

e.g. BLZ

SLT $at

BNE $at, 0, L

Java vs C: Pointers and Structures

Pointers are 32-bits, treat just like ints

Pointers to structs are pointers

C allows passing whole structs

- `int distance(struct Point p1, struct Point p2)`
- Treat like a collection of consecutive 32-bit arguments, use registers for first 4 words, stack for rest

- Of course, Inefficient and to be avoided, better to use `int distance(struct Point *p1, struct Point *p2)` in all cases

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; }

Global and Locals

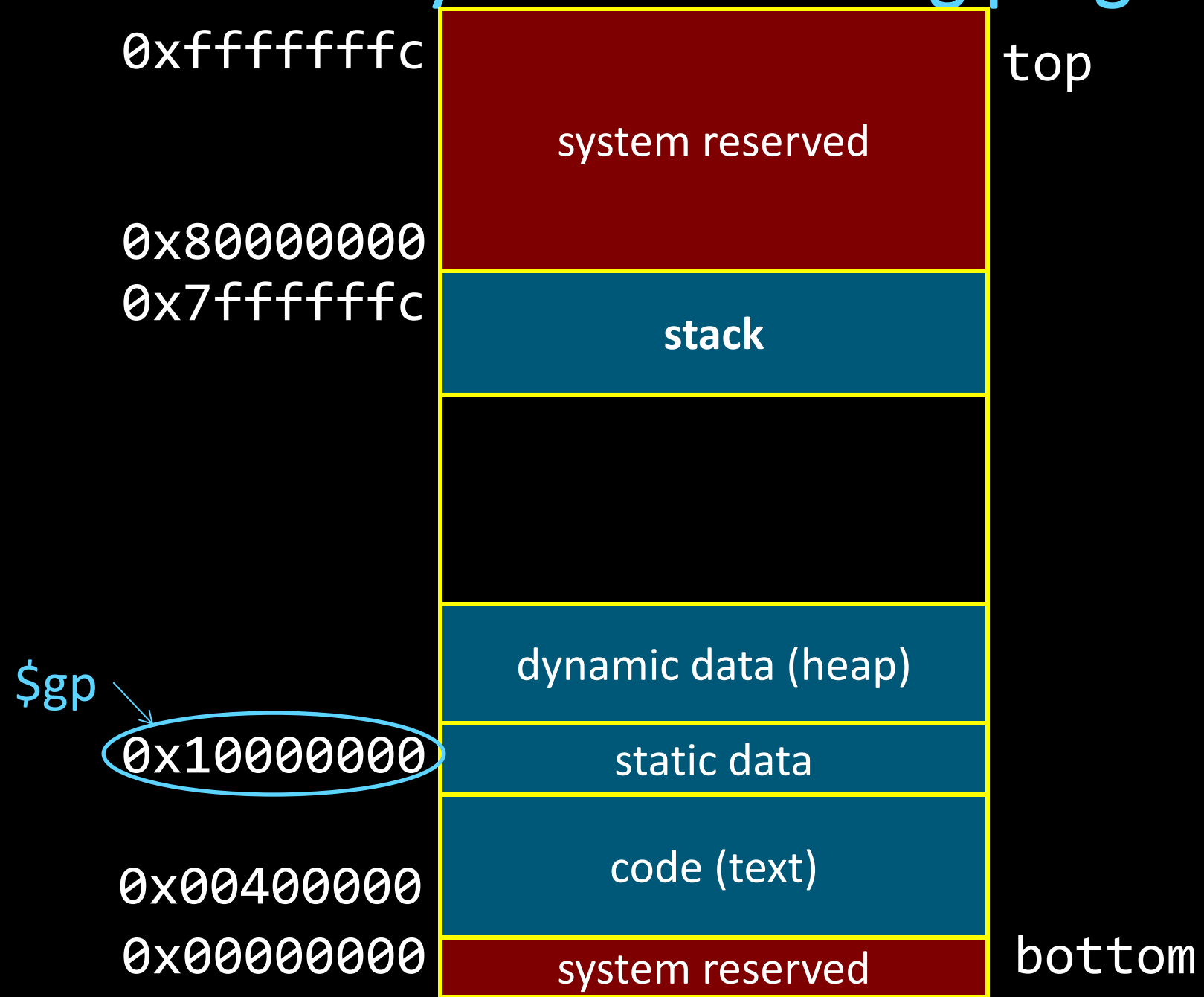
How does a function load global data?

- 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 top of the 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, $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 extra arguments to sub-functions
  - contains space for first 4 arguments to sub-functions

Next Goal

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

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

Callee-Save

```
main:
  addiu $sp,$sp,-32
  sw $31,28($sp)
  sw $30, 24($sp)
  sw $17, 20($sp)
  sw $16, 16($sp)
  addiu  $30, $sp, 28
            …
  [use $16 and $17]
            …
  lw $31,28($sp)
  lw $30,24($sp)
  lw $17, 20$sp)
  lw $16, 16($sp)
  addiu $sp,$sp,32
  jr $31
```

Assume caller is using the registers

Save on entry, restore on exit

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

Callee-Save

```
main:
  addiu $sp,$sp,-32
  sw $ra,28($sp)
  sw $fp, 24($sp)
  sw $s1, 20($sp)
  sw $s0, 16($sp)
  addiu  $fp, $sp, 28
            …
  [use $s0 and $s1]
            …
  lw $ra,28($sp)
  lw $fp,24($sp)
  lw $s1, 20$sp)
  lw $s0, 16($sp)
  addiu $sp,$sp,32
  jr $ra
```

Assume caller is using the registers

Save on entry, restore on exit

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

Caller-Save

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

Assume the registers are free for the taking, clobber them

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

By saving and restoring them before and after subroutine invocations

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

Caller-Save

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

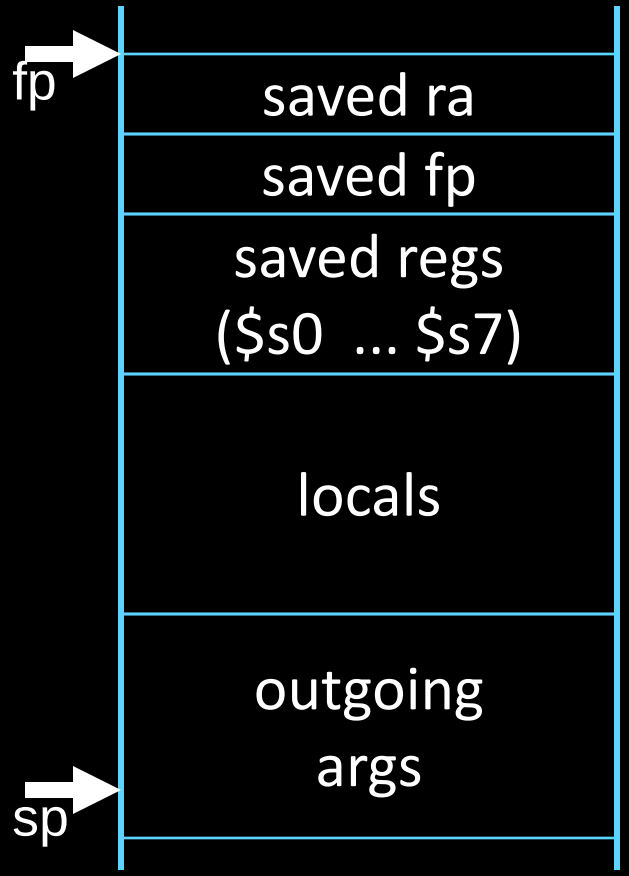
Assume the registers are free for the taking, clobber them

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

By saving and restoring them before and after subroutine invocations

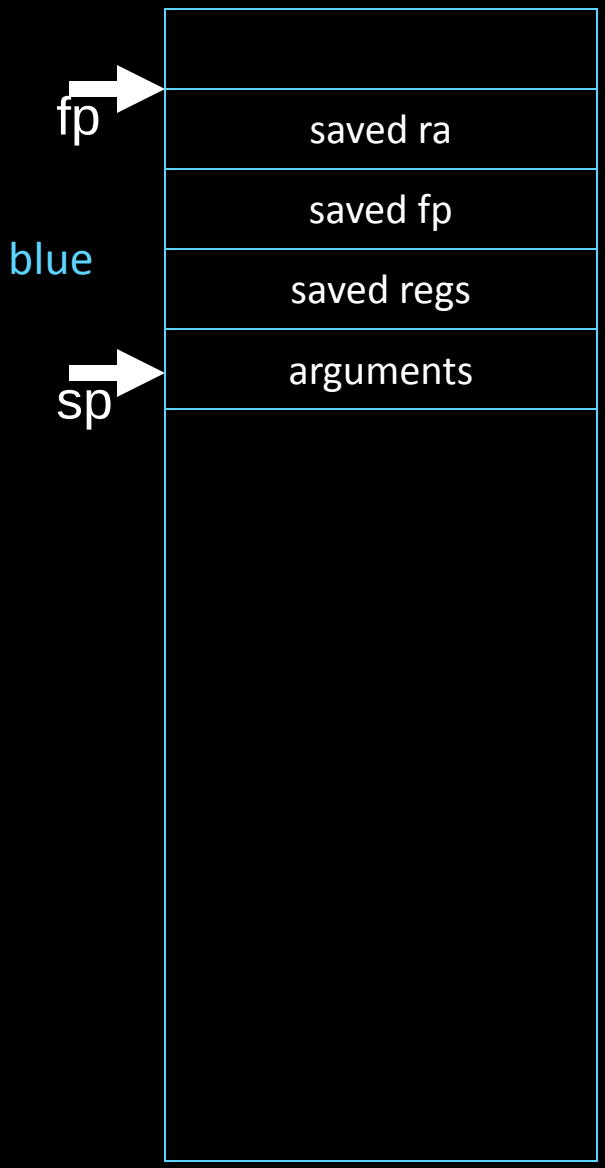
Pays off if a routine makes few calls to other routines with values that need to be preserved

Frame Layout on Stack



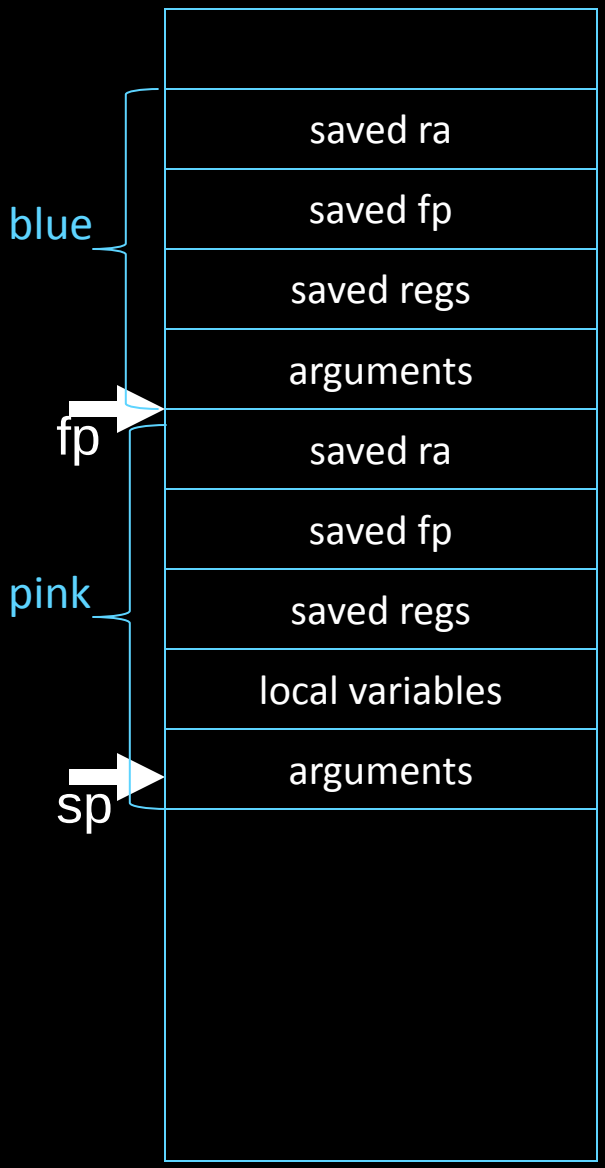

```
ADDIU $sp, $sp, -32 # allocate frame
SW $ra, 28($sp)     # save $ra
SW $fp, 24($sp)     # save old $fp
SW $s1, 20($sp)     # save ...
SW $s0, 16($sp)     # save ...
ADDIU $fp, $sp, 28  # set new frame ptr
...                 ...
BODY
...                 ...
LW $s0, 16($sp)     # restore …
LW $s1,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

Frame Layout on Stack



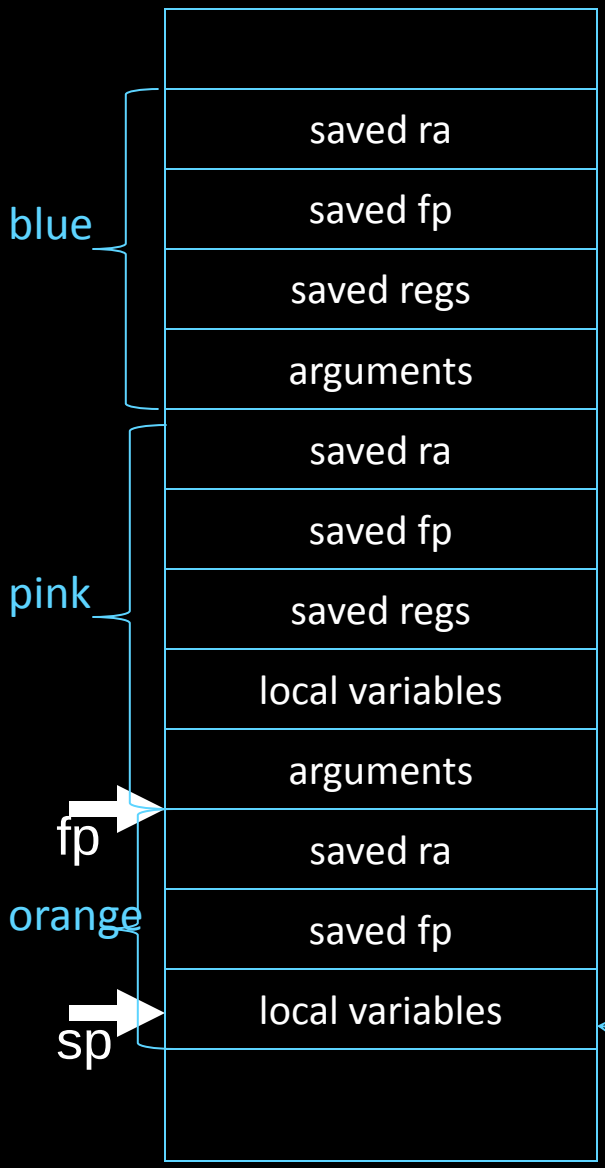

```
blue() {
   pink(0,1,2,3,4,5);
}
```

Frame Layout on Stack



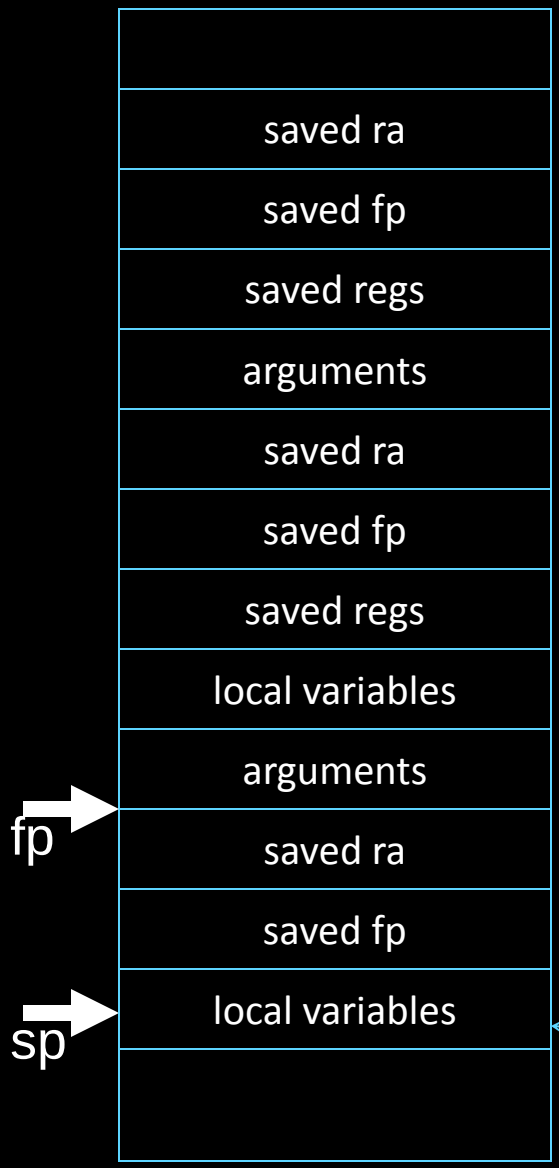

```
blue() {
   pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    orange(10,11,12,13,14);
}
```

Frame Layout on Stack




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

Buffer Overflow




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

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

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

Recap: Conventions so far

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent's stack frame
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains 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