CSCI 210: Computer Architecture
Lecture 12: Procedures

Stephen Checkoway
Oberlin College
Mar. 16, 2022
Slides from Cynthia Taylor

Announcements

- No class or office hours on Friday
- Problem Set 3 due Friday
- Problem Set 1 resubmit available _soon_
  - Due a week from when it's available
  - 25% of your grade comes from the original submission, 75% comes from the resubmission
- Lab 2 due Sunday
  - Make sure it runs on occs

Jump and Link

`jal Label`

- Address of following instruction put in $ra
- Jumps to target address
- Used for procedure calls

Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  – Address of following instruction put in $ra

  – Jumps to target address

- Procedure return: jump register

  `jr $ra`

  – Copies $ra to program counter

Recall: Procedures

```
int addTimes3(int x, int y){
  int w = y * 3;
  int z = x + w;
  return z;
}
```

Procedure Calling

1. Place arguments in registers: $a0, $a1, $a2, $a3
2. Transfer control to procedure: jal label
3. Acquire storage for procedure: use the stack
4. Perform procedure’s operations
5. Place result in register for caller: $v0, $v1
6. Return to place of call: jr $ra

What does a procedure call look like?

```
  …
  move  $a0, $s2
  jal   addTen
  # Now v0 holds the value of $s2 + 10
  …
addten:
  addi  $v0, $a0, 10
  jr    $ra
```

What is the problem with this code

```
move  $a0, $t2
move  $a1, $t3
jal   add
move  $t4, $v0
sub   $t4, $t4, $t2
```

```
#add $a0,$a1
add: add  $t2, $a0, $a1
     move $v0, $t2
     jr   $ra
```

A. Not adding correctly

B. $t2 is overwritten in add

C. We are not saving the return address before the procedure

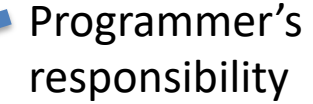
D. There is nothing wrong with this code

Register values across function calls

- “Preserved” registers
  - You can trust them to persist past function calls
    - Functions must ensure not to change them or to restore them if they do
- Not “Preserved” registers
  - Contents can be changed when you call a function
    - If you need the value, you need to put it somewhere else

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes



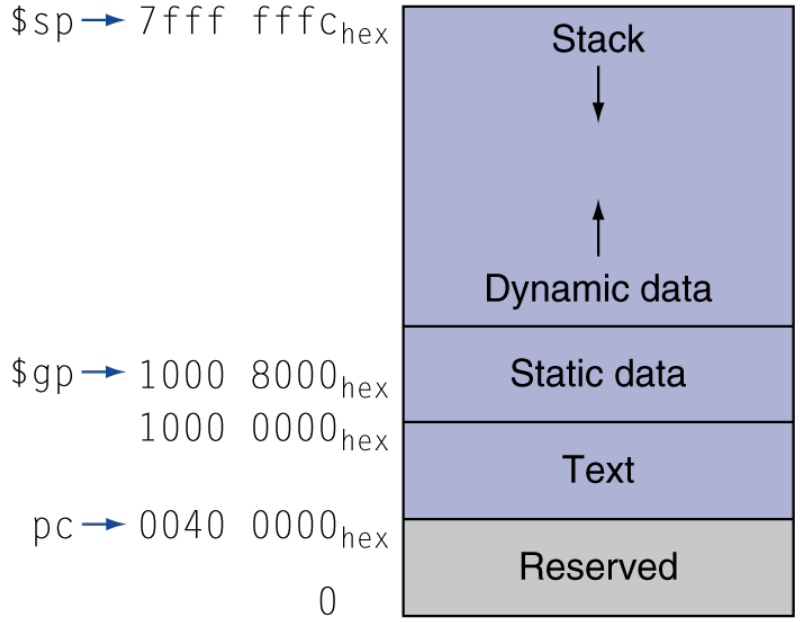
“Spill” and “Fill”

- Spill register to memory
  - Whenever you have too many variables to keep in registers
  - Whenever you call a method and need values in non-preserved registers
  - Whenever you want to use a preserved register and need to keep a copy

- Fill registers from memory
  - To restore previously spilled registers

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: “automatic” storage for procedures



Before and after a function

Assembly Code

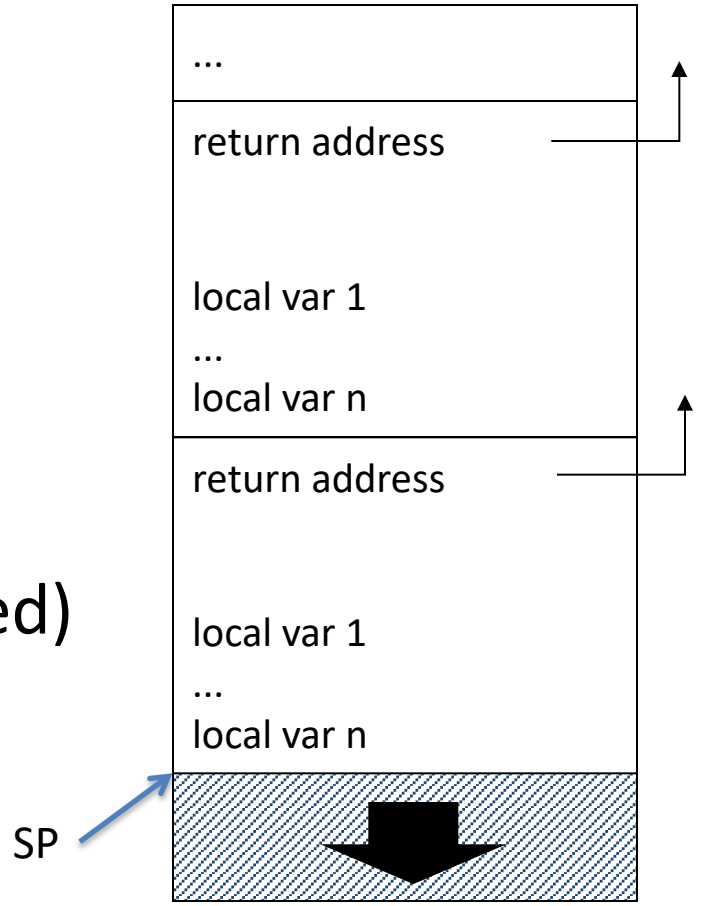
```
sw   $t0, 0($sp)
jal  myFunction
lw   $t0, 0($sp)
```

Which register is being spilled and filled?

A. $ra
B. $t0
C. $sp
D. No register is spilled/filled
E. No need to spill/fill any registers

Stack

- Stack of stack frames
  - One per pending procedure
- Each stack frame stores
  - Where to return to
  - Local variables
  - Arguments for called functions (if needed)
- Stack pointer points to last record

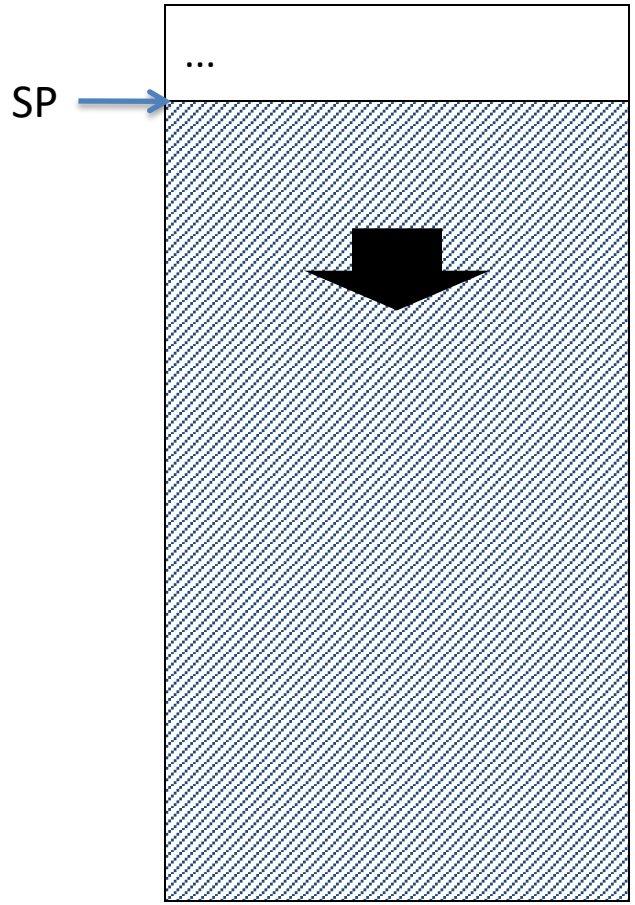


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
  int n = 10;
  bar(n);
  return n;
}

bar(int n) {
   return n + 2;
}
```

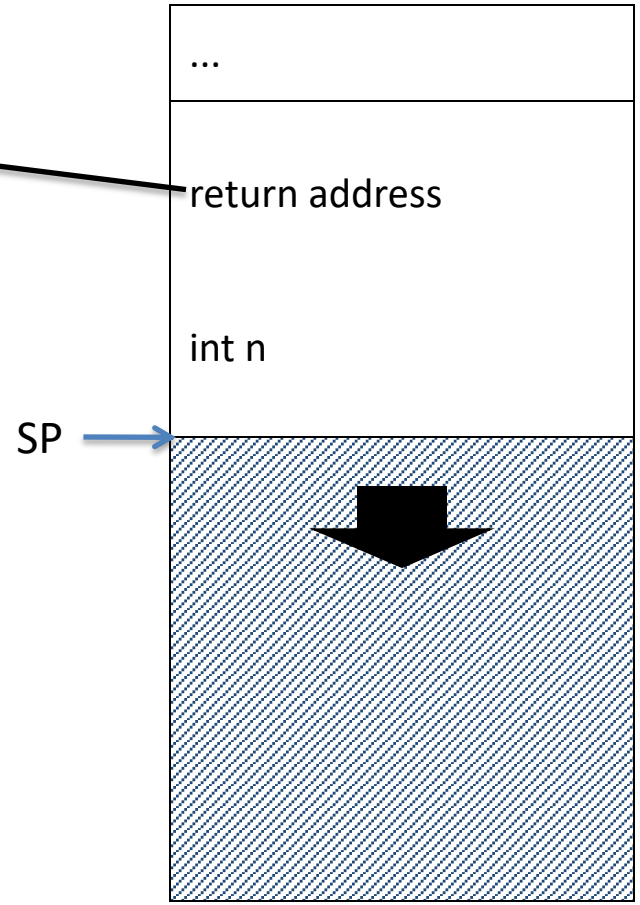


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
   int n = 10;
   bar(n);
   return n;
}

bar(int n) {
   return n + 2;
}
```

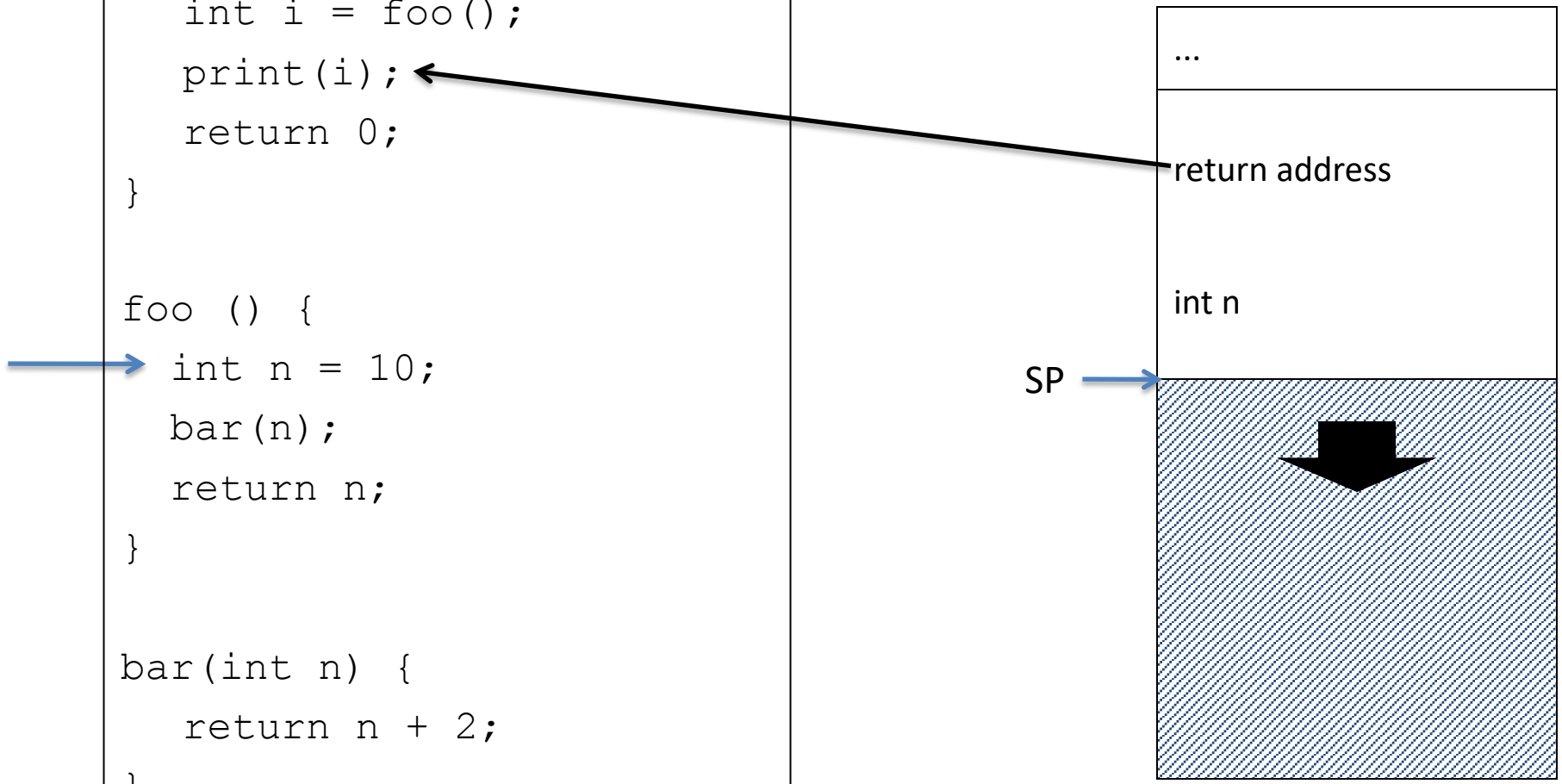


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
   int n = 10;
   bar(n);
   return n;
}

bar(int n) {
   return n + 2;
}
```

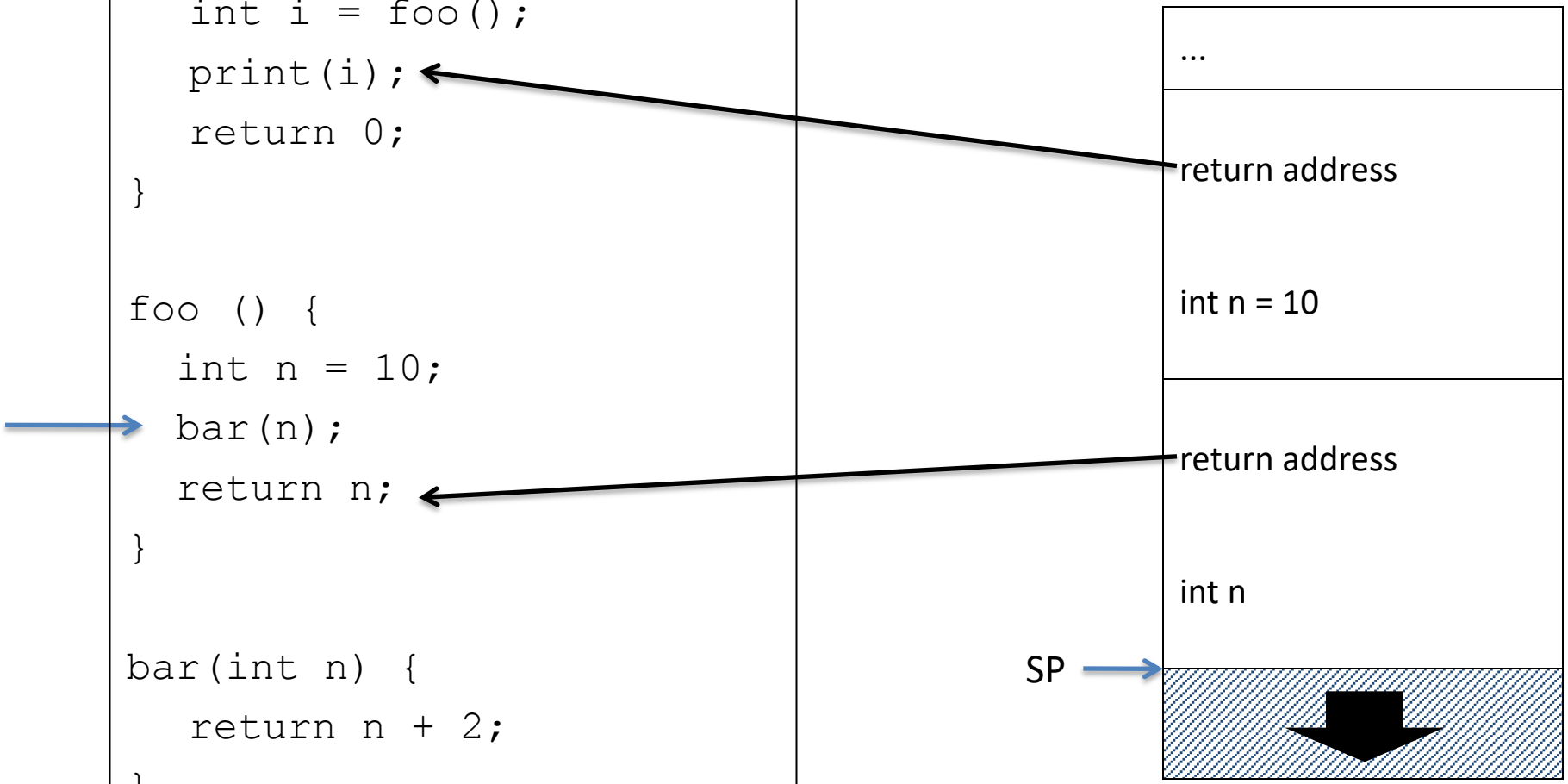


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
   int n = 10;
   bar(n);
   return n;
}

bar(int n) {
   return n + 2;
}
```

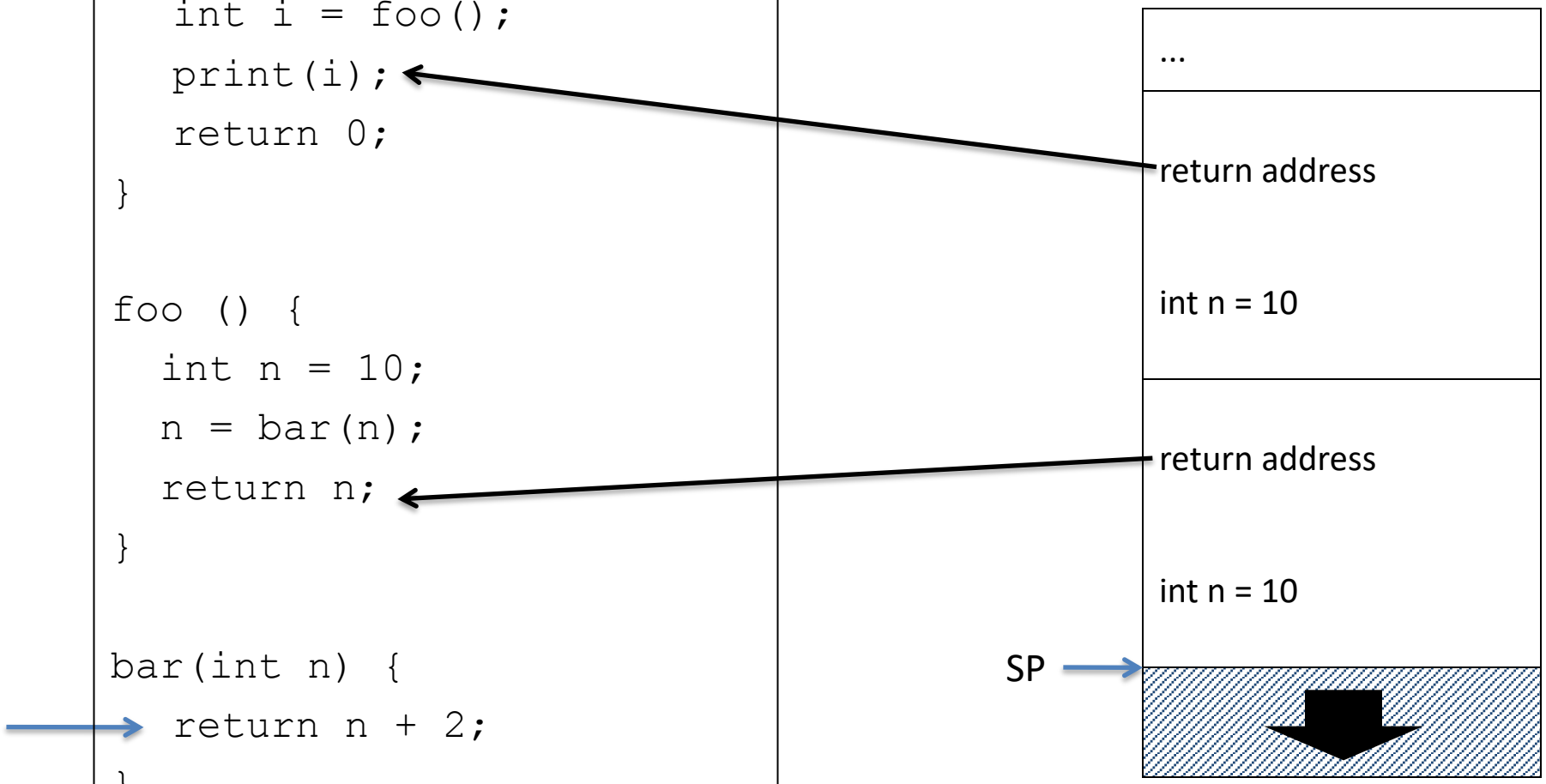


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
  int n = 10;
  n = bar(n);
  return n;
}

bar(int n) {
   return n + 2;
}
```

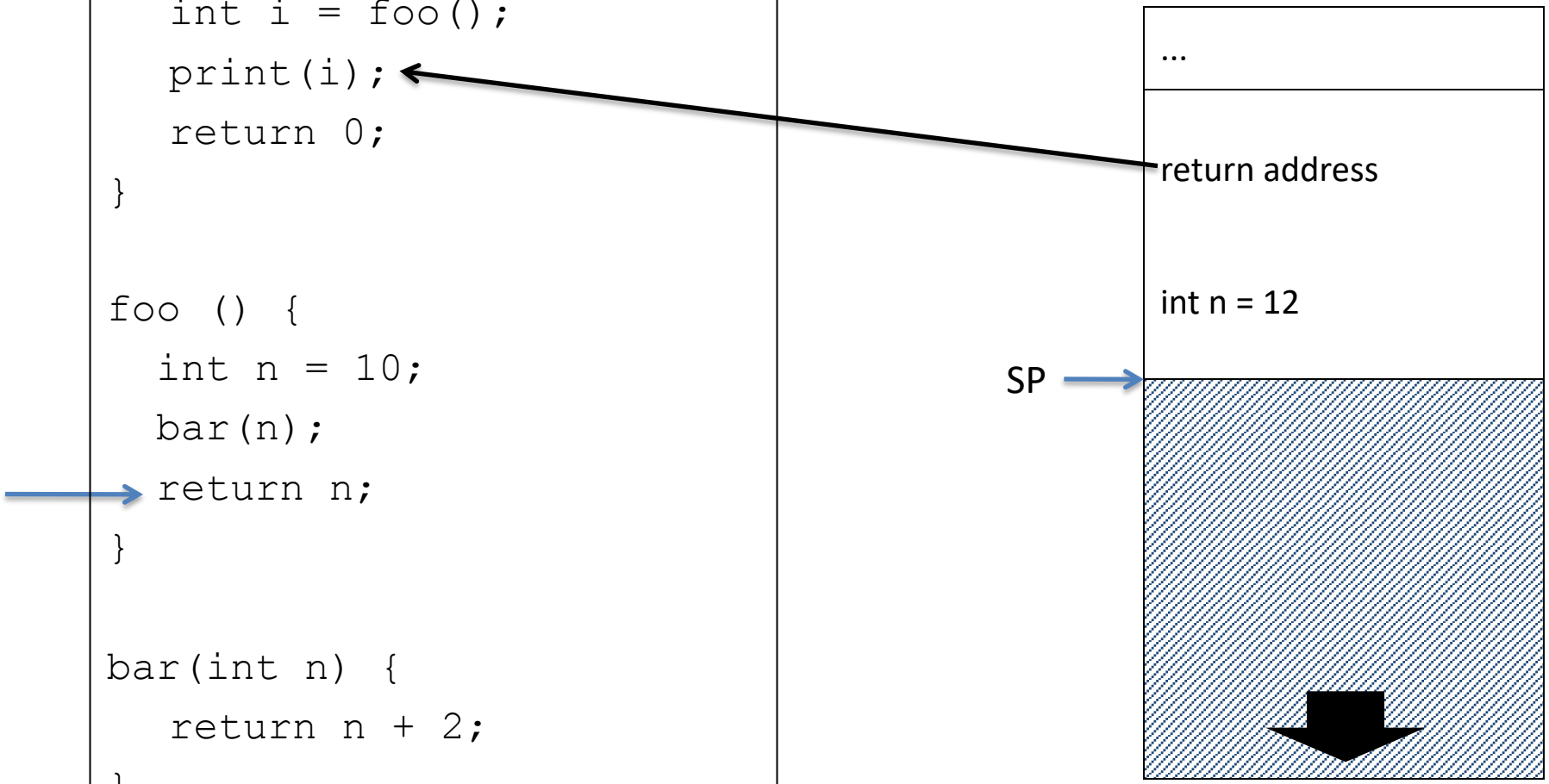


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
   int n = 10;
   bar(n);
   return n;
}

bar(int n) {
   return n + 2;
}
```

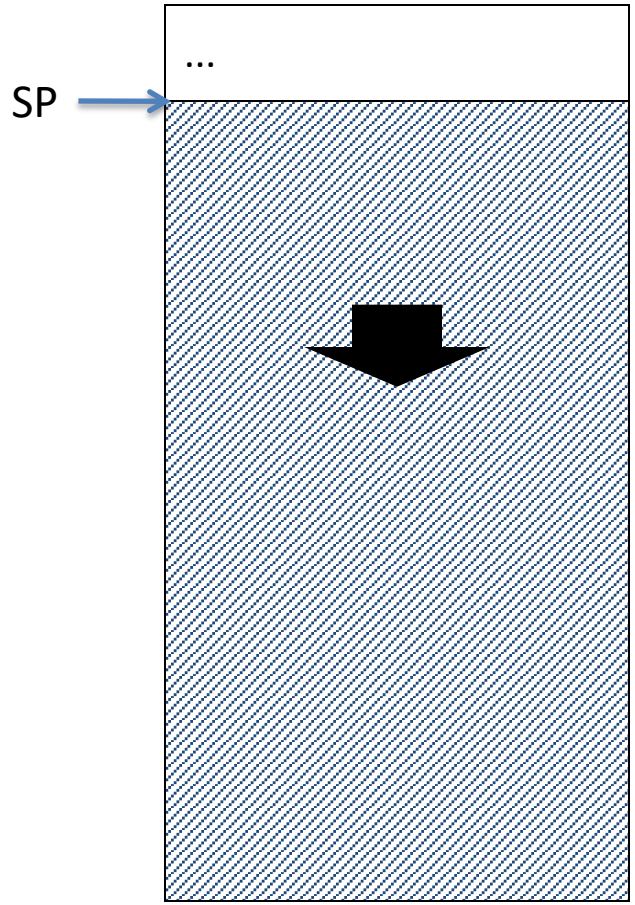


Process Stack

```
main () {
   int i = foo();
   print(i);
   return 0;
}

foo () {
  int n = 10;
  bar(n);
  return n;
}

bar(int n) {
   return n + 2;
}
```



To add a variable to the stack in MIPS

- Change the stack pointer $sp to create room on the stack for the variable

- Use sw to store the variable on the stack

Stack

If you wish to push an integer variable to the top of the stack, which of the following is true:

A. You should decrement the stack pointer ($sp) by 1

B. You should decrement $sp by 4

C. You should increment $sp by 1

D. You should increment $sp by 4

E. None of the above

- To add the contents of $s0 to the stack
  - addi $sp, $sp, -4
    sw $s0, 0($sp)
- To get the value back from the stack
  - lw $s0, 0($sp)
- To “erase” the value from the stack
  - addi $sp, $sp, 4

Spill and fill the return address; why?

```
addi $sp, $sp, -4
sw   $ra, 0($sp)
jal  myFunction
lw   $ra, 0($sp)
addi $sp, $sp, 4
```

A better approach

- In the function “prologue,” reserve space on the stack for all of the variables and saved registers you’ll need
- Use sw/lw to spill and fill as needed to the space reserved in the prologue
- In the function “epilogue,” restore any saved registers you need and update the stack pointer

Complete example

```
foo:
        addi    $sp, $sp, -12   # Reserve space for 3 vars
        sw      $ra, 8($sp)     # Stores (spills) $ra, return address
        sw      $s0, 4($sp)     # Stores (spills) s0, callee-saved reg
        …
        li      $s0, 25         # Set s0 to 25
        sw      $t3, 0($sp)     # Stores (spills) t3, caller-saved reg
        add     $a0, $t1, $t3
        jal     myFunction
        lw      $t3, 0($sp)     # Restores (fills) t3
        …
        lw      $s0,  4($sp)    # Restores (fills) s0, must restore
        lw      $ra,  8($sp)    # Restores (fills) $ra, return address
        addi    $sp, $sp, 12    # Restore the stack pointer
        jr      $ra             # Return
```

Leaf function

- If the function doesn't call any other functions, it's a "leaf"
- If a leaf function doesn't need to use any of the callee-saved registers (e.g., $s0–$s7), then it doesn't need to change the stack pointer or spill/fill $ra
- Example:

```
# myFunction(int a0, int a1, int a2)
myFunction:
        add    $t0, $a0, $a2
        sub    $v0, $t0, $a1
        jr     $ra
```

Reading

- Next lecture: More stack!
- Problem Set 3 due Friday
- Lab 2 due Sunday