

# CSCI 210: Computer Architecture

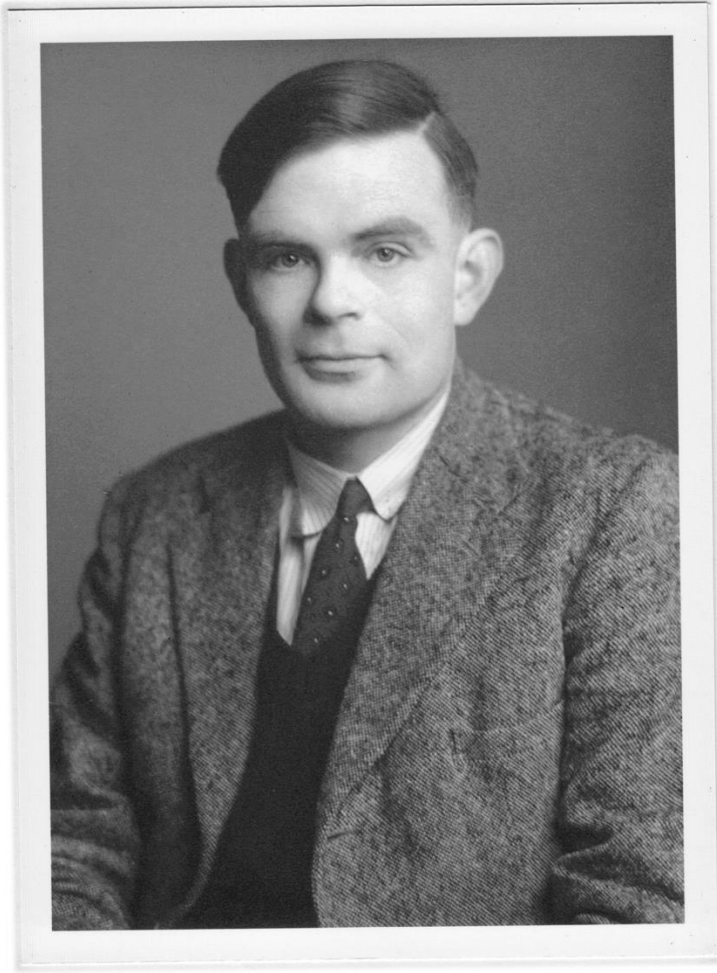
## Lecture 11: Procedures

Stephen Checkoway  
Slides from Cynthia Taylor

# Announcements

- Problem Set 3 Due Friday
- Lab 2 due Monday

# CS History: The Subroutine



- A group of instructions we can re-run as a unit
- Conceived of by Alan Turing in 1945, independently implemented by Kay McNulty and others on the ENIAC in 1947, formally developed by Maurice Wilkes, David Wheeler, and Stanley Gill in 1952.
- In early computers, loaded as strips of paper tape or collections of punch cards that would be reinserted into the machine
- Later developed as macros, pieces of code the assembler would copy into multiple places during assembly

# Recall from Last Class

- Fetch/Decode/Execute cycle
  - $IR = \text{Memory}[PC]$
  - $PC = PC + 4$
- Branch instructions change PC value conditionally
  - `beq`, `bne`
  - Used with `slt`
- Jump instructions always change PC value
  - `j`, `jr`

# Jump

- `j label`
  - Go directly to the label (i.e.  $PC = \text{label}$ )
- `jr register`
  - Go directly to the address specified in the register

C Code

```
if (X == 0)
    X = Y + Z;
else
    X = Z + Z;
```

Assuming X, Y, and Z are integers in registers \$t0, \$t1, and \$t2, respectively, which are the equivalent assembly instructions?

```
bne $t0, $zero, false
add $t0, $t1, $t2
false: add $t0, $t2, $t2
```

A

```
bne $t0, $zero, false
add $t0, $t1, $t2
j endif
false: add $t0, $t2, $t2
endif:
```

B

```
bne $t0, $zero, false
j endif
add $t0, $t1, $t2
false: add $t0, $t2, $t2
endif:
```

C

D – None of the above

## C Code

```
while (i < 10) {  
    i = i + 1;  
}
```

Assume i is in \$t0. What is the equivalent assembly?

slti rd, rs, imm  
if (rs < imm) rd = 1; else rd = 0;

```
w: slti $t2, $t0, 10  
    beq $t2, $zero, end  
    addi $t0, $t0, 1  
    j w  
end:
```

A

```
w: slti $t2, $t0, 10  
    beq $t2, $zero, end  
    addi $t0, $t0, 1  
end:
```

B

```
    slti $t2, $t0, 10  
w: beq $t2, $zero, end  
    addi $t0, $t0, 1  
    j w  
end:
```

C

D – More than one of these

E – None of these

# How to access an array in a for loop

- Can't programmatically change the offset
- Need to change the *base address* instead
- Add 4 to the base address every time you want to move up an element of the array



```
for (i=0; i < 10; i++) {  
    A[i] = 0;  
}
```

\*Assume base address of A is in \$s3

```
    add $s0, $zero, $zero  
    addi $s1, $zero, 40  
Loop: beq $s0, $s1, End  
    add $s4, $s3, $s0  
    sw $zero, 0($s4)  
    addi $s0, $s0, 4  
    j Loop  
End:
```

## C Code

```
for (i = 0; i < 10; i++) {  
    A[i+1] = A[i];  
}
```

Assume the base address of A is in \$t0, and i is in \$t1. Each element of A is 4 bytes. What is the equivalent assembly?

```
    addi $t2, $zero, 10  
    add $t1, $zero, $zero  
for: bne $t1, $t2, end  
    lw $t3, $t1($t0)  
    addi $t1, $t1, 1  
    sw $t3, $t1($t0)  
    j for  
end:
```

A

```
    addi $t2, $zero, 40  
    add $t1, $zero, $zero  
for: beq $t1, $t2, end  
    add $t4, $t0, $t1  
    lw $t3, 0($t4)  
    addi $t1, $t1, 4  
    add $t4, $t0, $t1  
    sw $t3, 0($t4)  
    j for  
end:
```

B

```
    addi $t2, $zero, 10  
    add $t1, $zero, $zero  
    bne $t1, $t2, end  
    add $t4, $t0, $t1  
    lw $t3, 0($t4)  
    addi $t1, $t1, 1  
    add $t4, $t0, $t1  
    sw $t3, 0($t4)  
end:
```

C

D – More than one of these

E – None of these

# Jump and Link

`jal Label`

- Address of following instruction put in `$ra`
- Jumps to target address given by label

What is the most common use of a jal instruction and why?

	Most common use	Best answer
A	Procedure call	Jal stores the next instruction in your current function so the called function knows where to return to.
B	Procedure call	Jal enables a long jump and most procedures are a fairly long distance away
C	If/else	Jal lets you go to the if while storing pc+4 (else)
D	If/else	Jal enables a long branch and most if statements are a fairly long distance away
E	None of the above	

# 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?

addten:

```
addi    $v0, $a0, 10
```

```
jr      $ra
```

...

```
move    $a0, $s2
```

```
jal     addTen
```

```
# Now v0 holds $s2 + 10
```

...



# 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 ( <b>hardware</b> )	n.a.
\$at	1	<b>reserved</b> 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	<b>yes</b>
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	<b>yes</b>
\$sp	29	stack pointer	<b>yes</b>
\$fp	30	frame pointer	<b>yes</b>
\$ra	31	return addr ( <b>hardware</b> )	<b>yes</b>

Programmer's responsibility

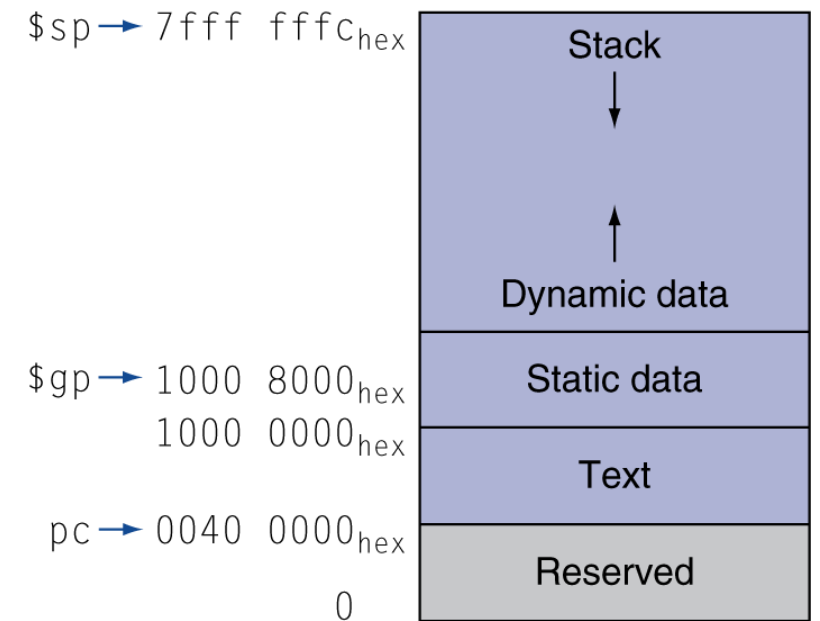


# “Spill” and “Fill”

- 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
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: “automatic” storage for procedures



# Before and after a function

Assembly Code

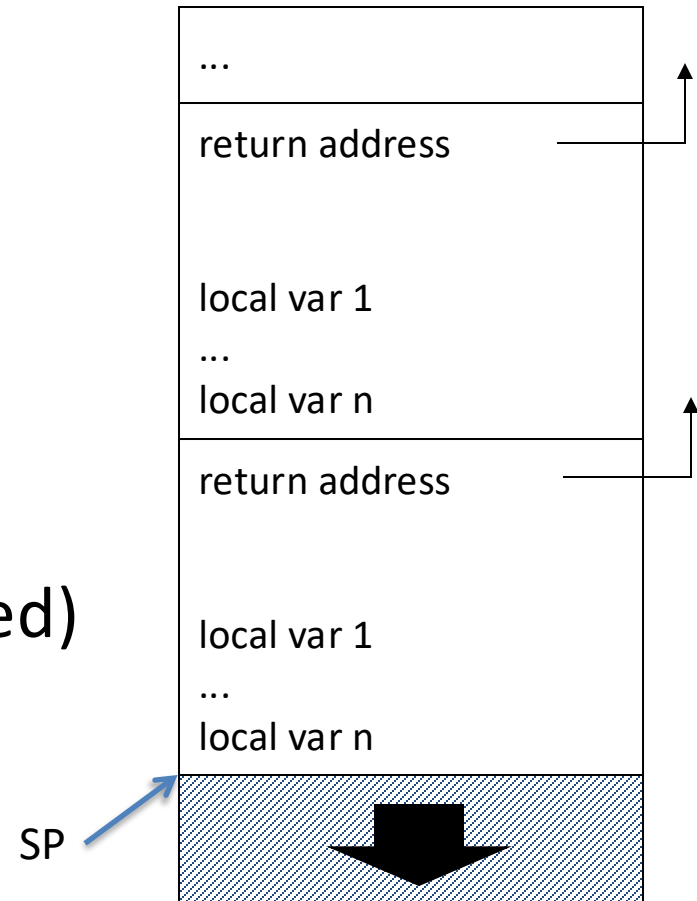
```
sw    $t0, 4($sp)
jal   myFunction
lw    $t0, 4($sp)
```

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

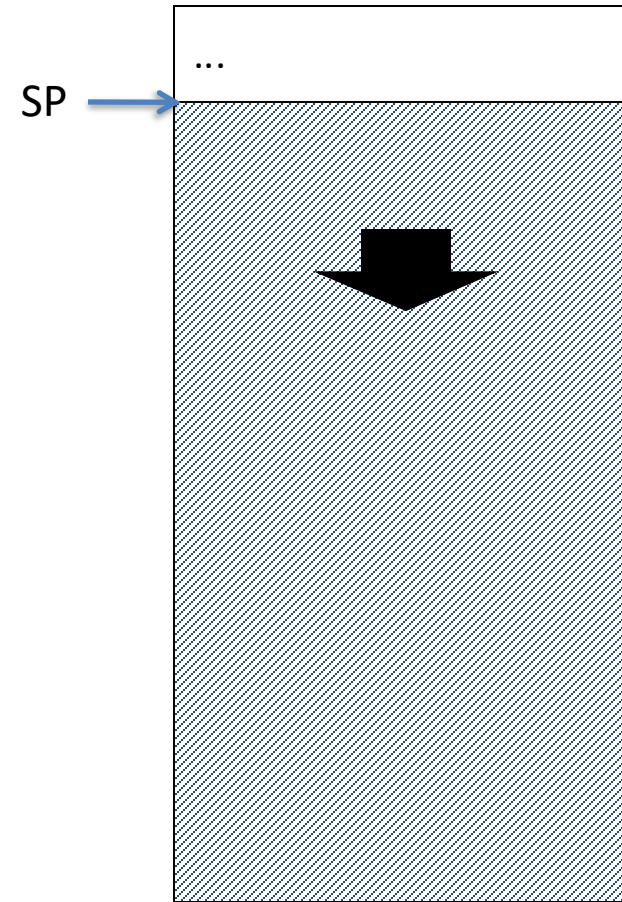
# Stack

- 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



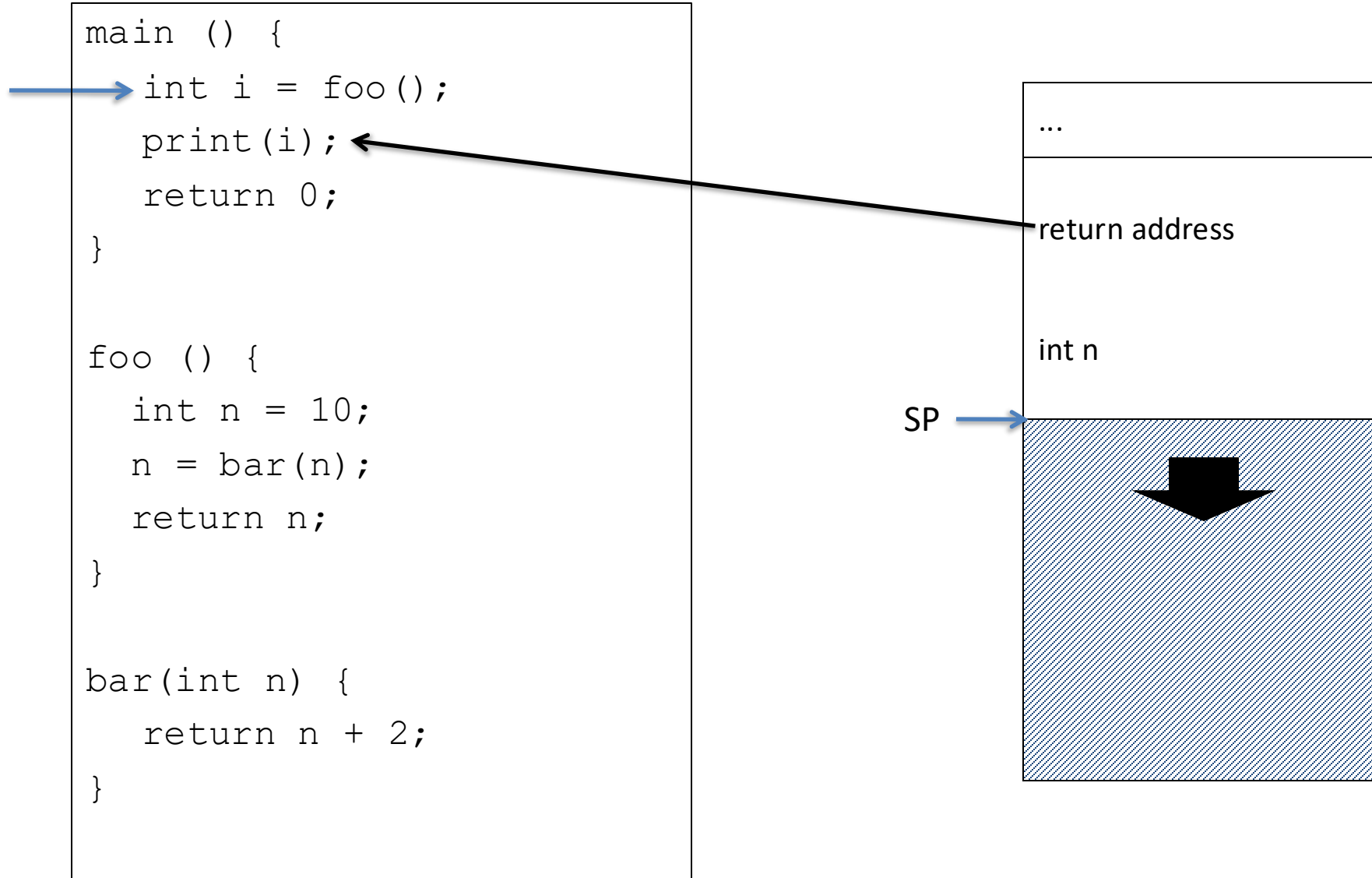
# Process Stack

```
main () {  
    int i = foo();  
    print(i);  
    return 0;  
}  
  
foo () {  
    int n = 10;  
    n = bar(n);  
    return n;  
}  
  
bar(int n) {  
    return n + 2;  
}
```

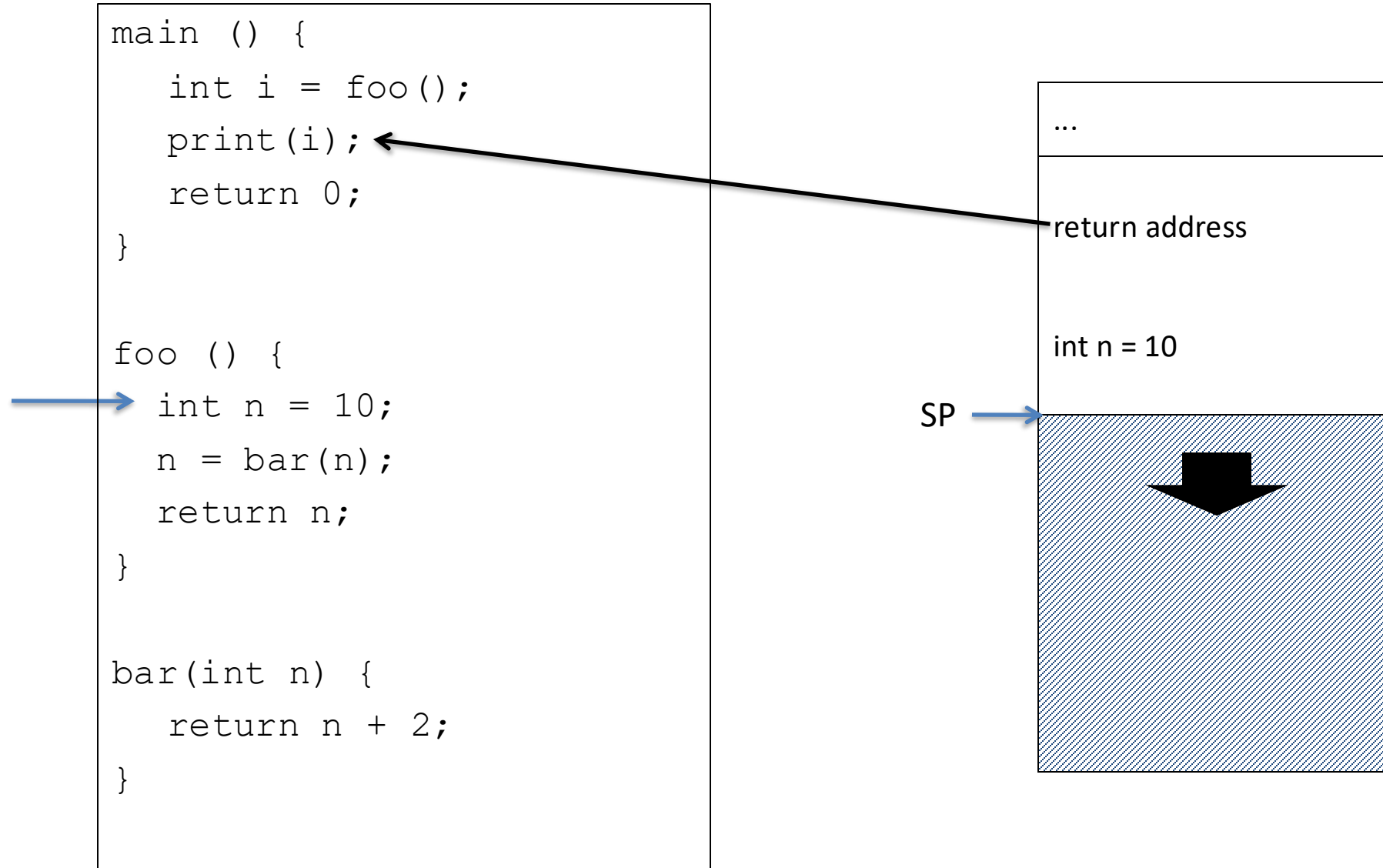




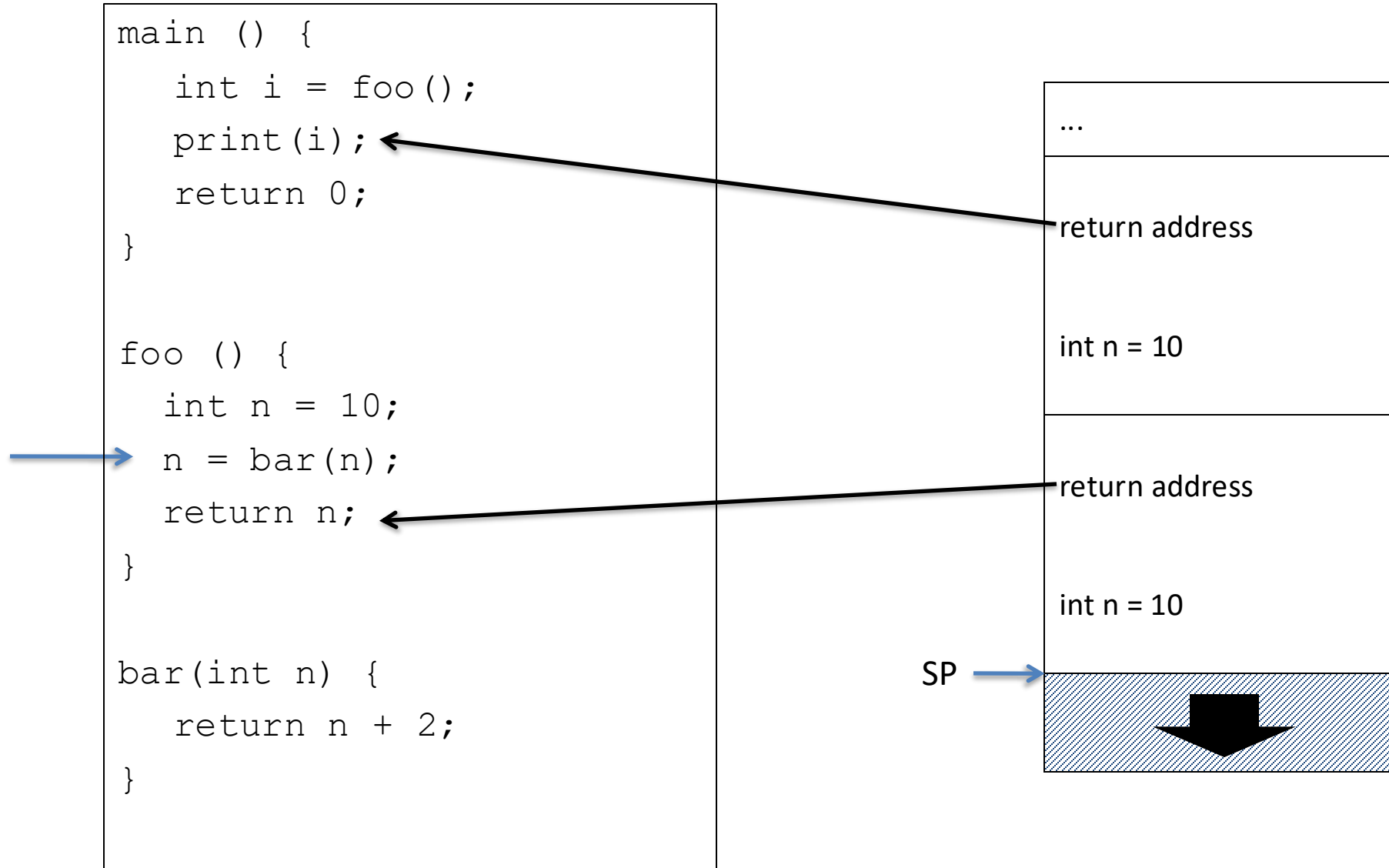
# Process Stack



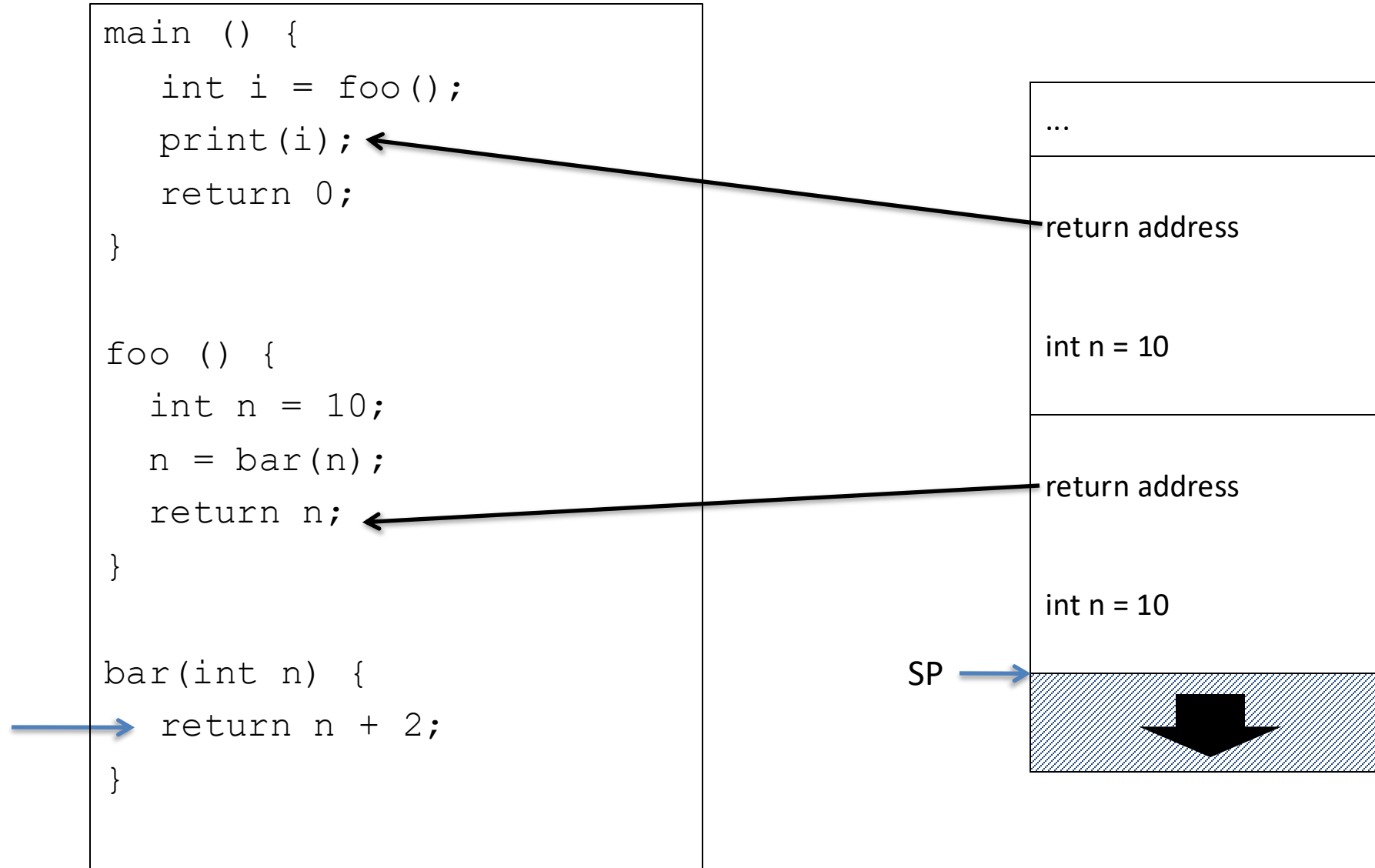
# Process Stack



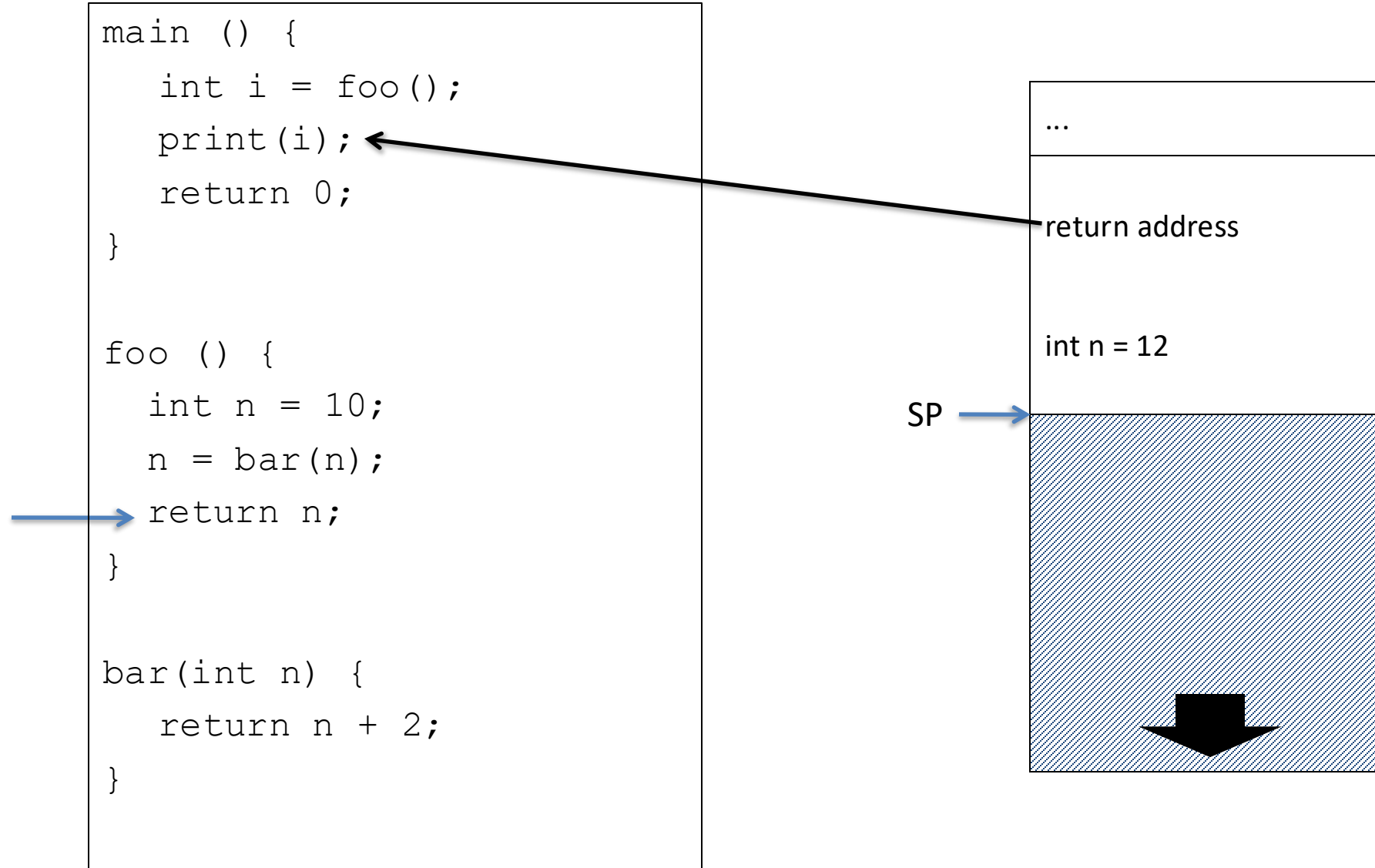
# Process Stack



# Process Stack

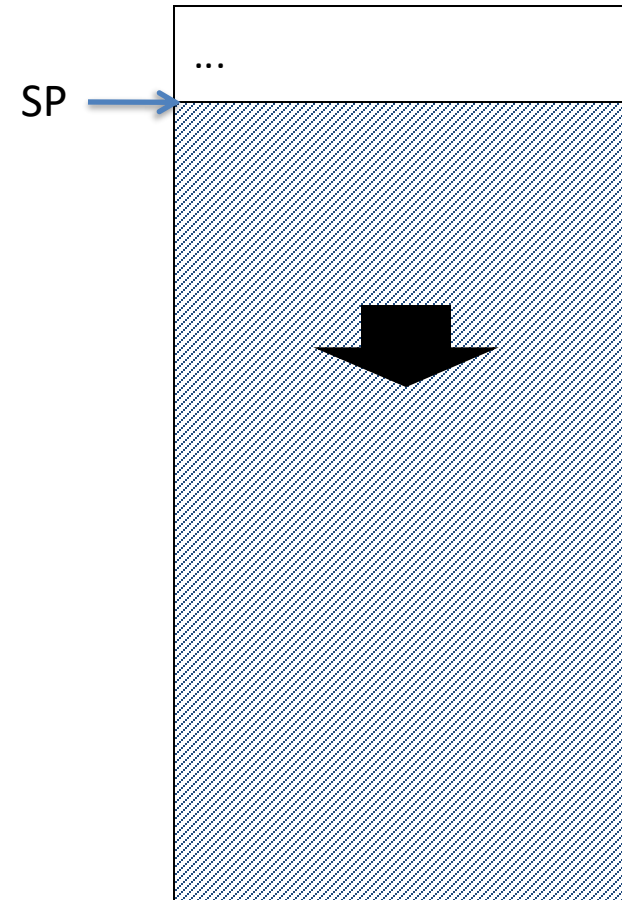


# Process Stack



# 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;  
}
```



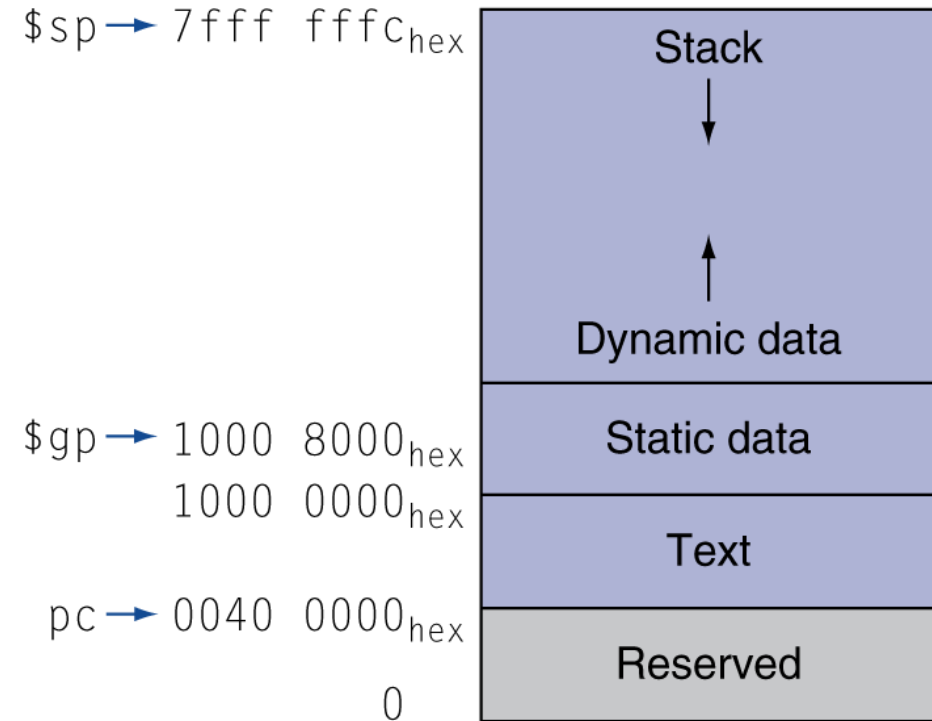
# To add a variable to the stack in MIPS

- 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





# Manipulating the Stack

- 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

# Think-Pair-Share: Why do we spill and fill the return address when we call a function from inside another function?

```
func1:
    . . .
    addi $sp, $sp, -4
    sw   $ra, 0($sp)
    jal  func2
    lw   $ra, 0($sp)
    addi $sp, $sp, 4
    . . .
    jr   $ra
```

# 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
```

# Reading

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