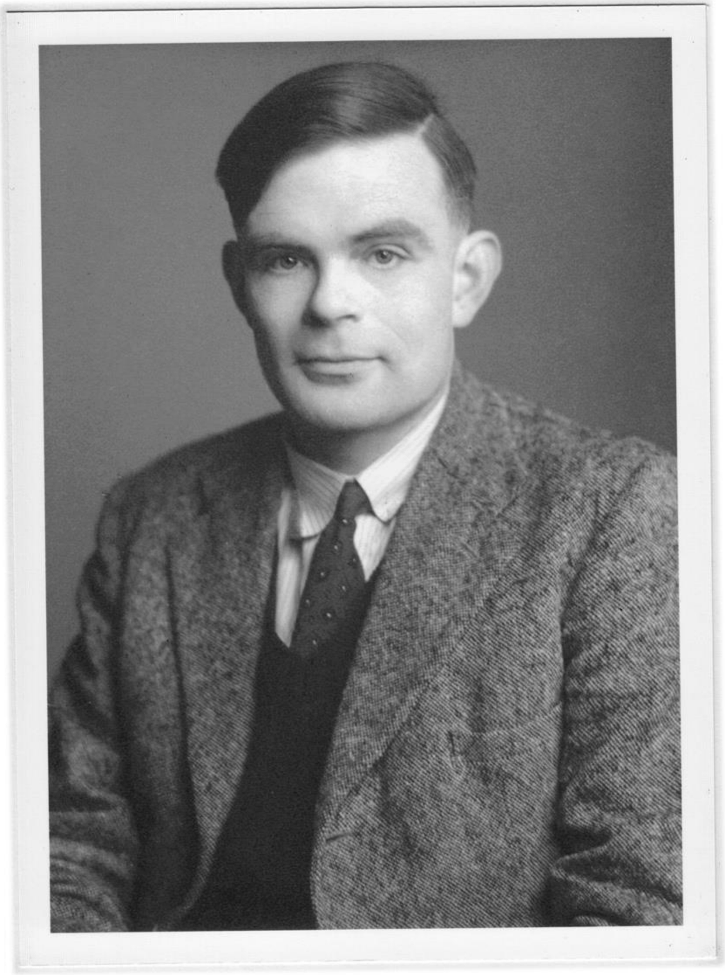
CSCI 210: Computer Architecture
Lecture 11: Procedures

Stephen Checkoway

Slides from Cynthia Taylor

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

Subroutines/functions: A high-level view

- Code in programs is organized in functions
- Functions take arguments
- Functions can call functions, including themselves
- Functions have local variables that are not shared with other functions, including other invocations of the same function (i.e., recursive calls to a function have different local variables)
- Functions return to the function that called them

Implications

- Functions take arguments: Need a way to access arguments
- Functions can call functions: Need a way to pass arguments
- Functions have local variables: Need per-function-call memory to hold the variables
- Functions return: Need to know what the return address is

Activation Records

- A per-function-call data structure that holds
  - Local variables/temporary storage space
  - Return address
  - Saved register values
  - Arguments for the next function call

Stack of activation records

- Each time a function is called, a new activation record is pushed onto a stack
- Each time a function returns, the activation record is popped off the stack

main() calls print_string() which calls print_char()

main	Local variables/temporary space for main
	Return address: where main returns
	Saved register values
	Arguments for print_string call
print_string	Local variables/temporary space for print_string
	Return address: where in main to returns
	Saved register values
	Arguments for print_char call
print_char	Local variables/temporary space for print_string
	Return address: where in print_char to return
	Saved register values
	Arguments for functions called by print_char

From theory to practice

- Activation record is the name we give to the data structure
- A **stack frame** is how an activation record is realized in software

Figure 3-21: Stack Frame

Base	Offset	Contents	Frame
		unspecified . . . variable size	*High addresses*
	+16	(if present) incoming arguments passed in stack frame	Previous
old *$sp*	+0	space for incoming arguments 1-4	
		locals and temporaries	
		general register save area	Current
		floating-point register save area	
$sp	+0	argument build area	*Low addresses*

Recall from Last Class

- Fetch/Decode/Execute cycle
  - IR = Memory[PC]
  - PC = PC + 4

- Branch instructions change PC value conditionally
  - `beq, bne`
  - Used with `slt`

- Jump instructions always change PC value
  - `j, jal, jr`

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

Procedure Calling

1. Place arguments in registers: $a0, $a1, $a2, $a3
2. Transfer control to procedure: jal label
3. Allocate stack frame for procedure (when necessary)
4. Perform procedure's operations
5. Place result in register for caller: $v0
6. Deallocate the stack frame (when allocated)
7. 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, if anything, is wrong 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 not 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

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

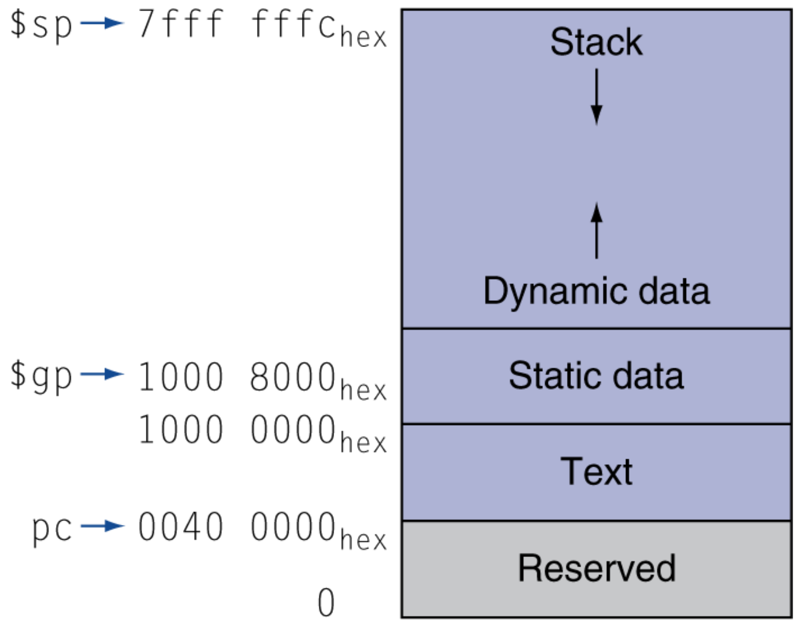
Programmer's responsibility

“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

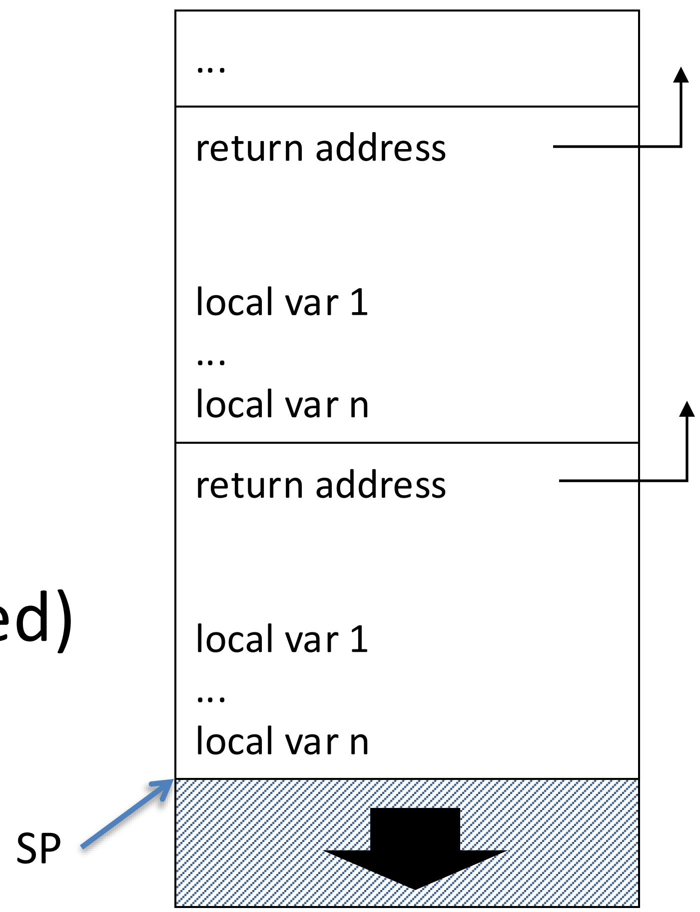
```
sw   $t0, 20($sp)
jal  myFunction
lw   $t0, 20($sp)
```

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

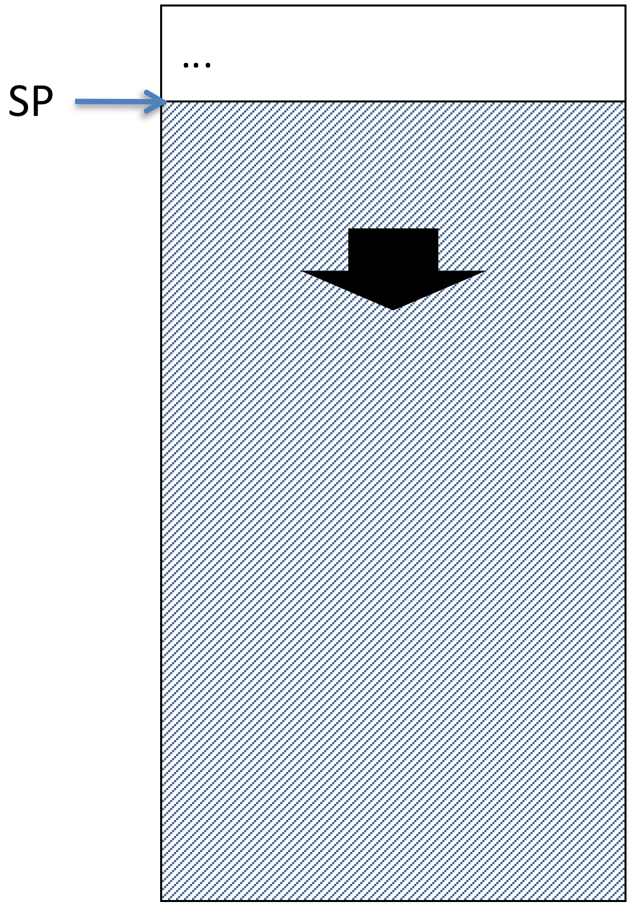


Process Stack

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

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

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

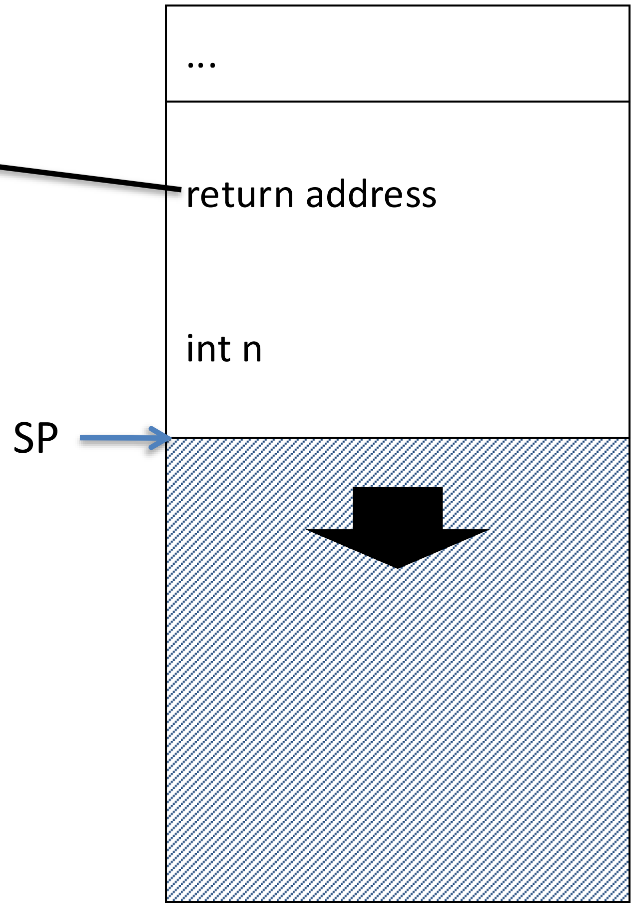


Process Stack

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

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

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

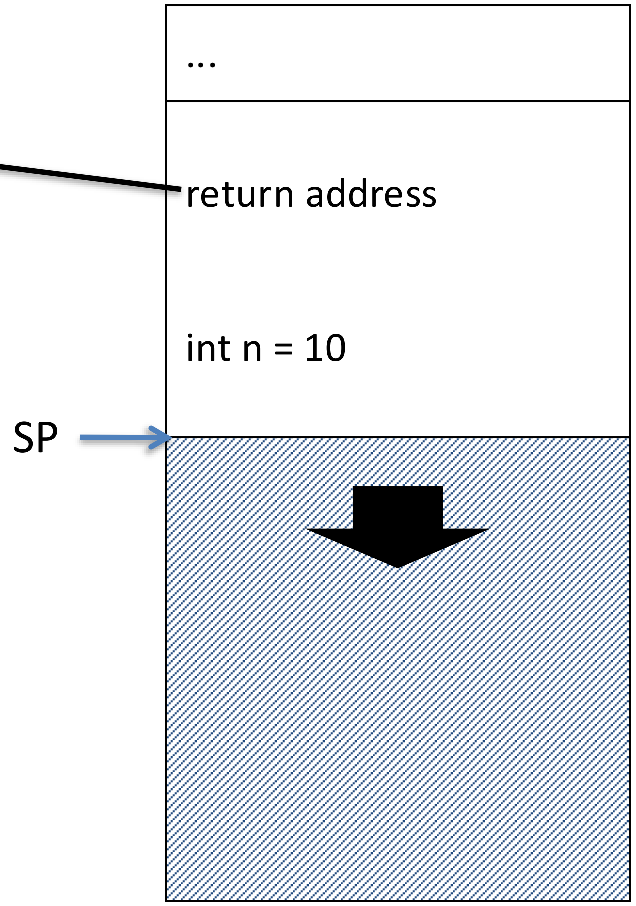


Process Stack

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

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

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

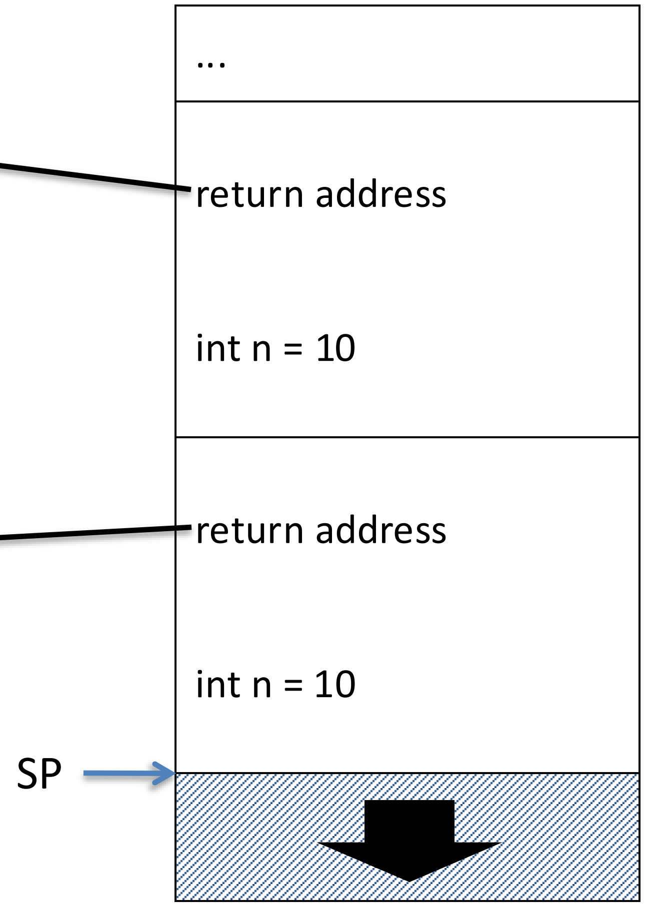


Process Stack

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

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

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

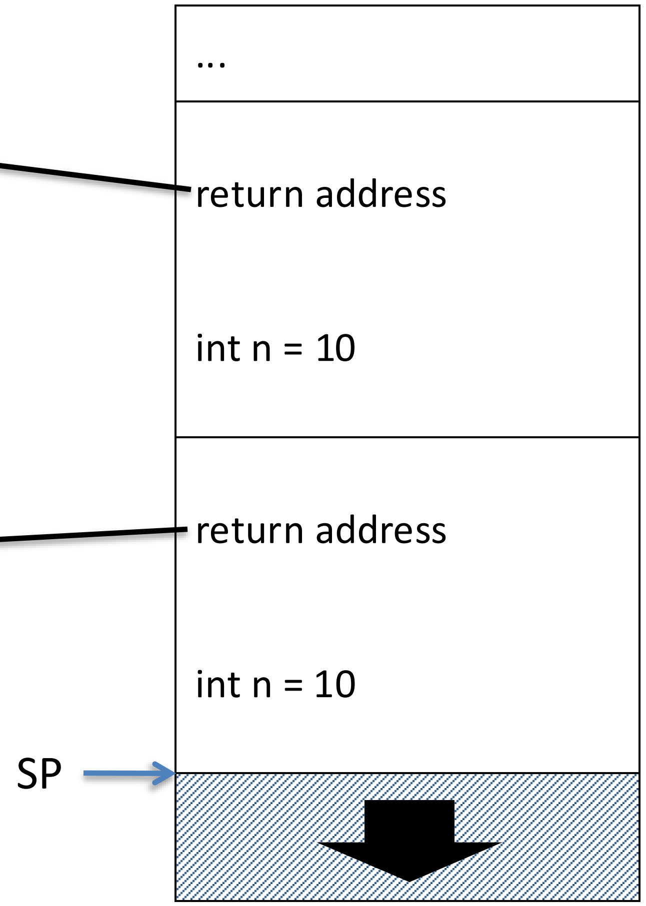


Process Stack

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

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

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

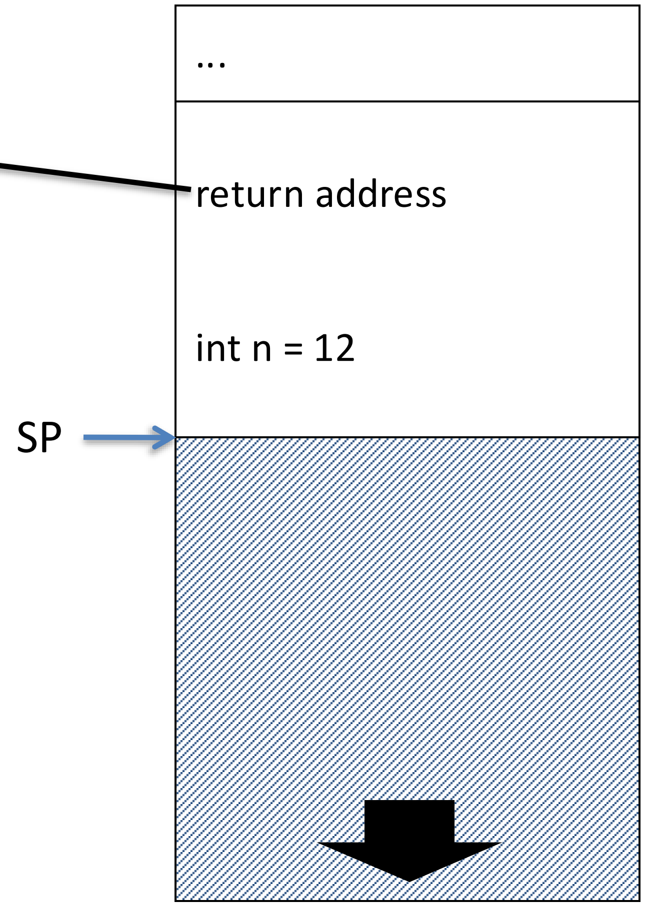


Process Stack

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

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

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

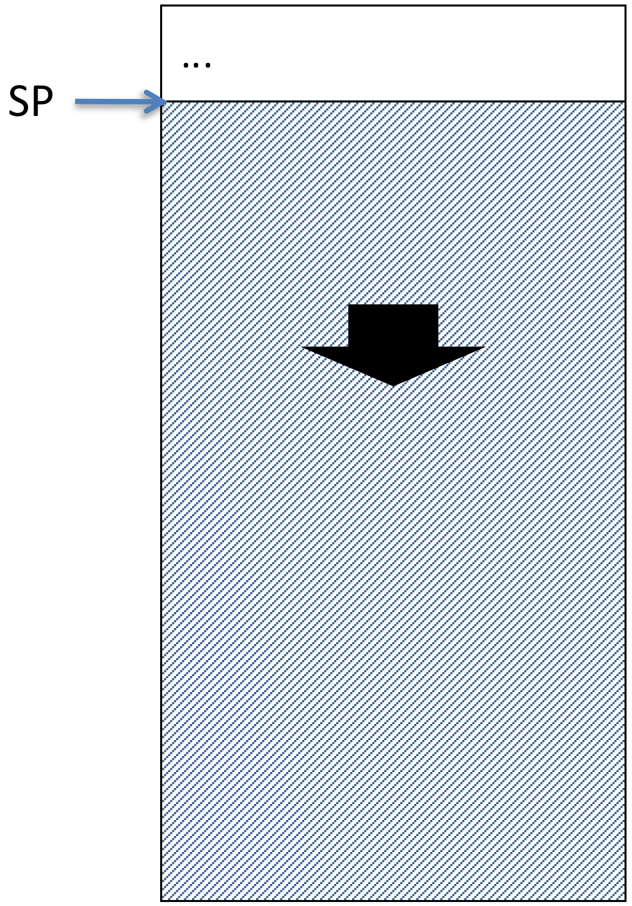


Process Stack

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

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

int 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:

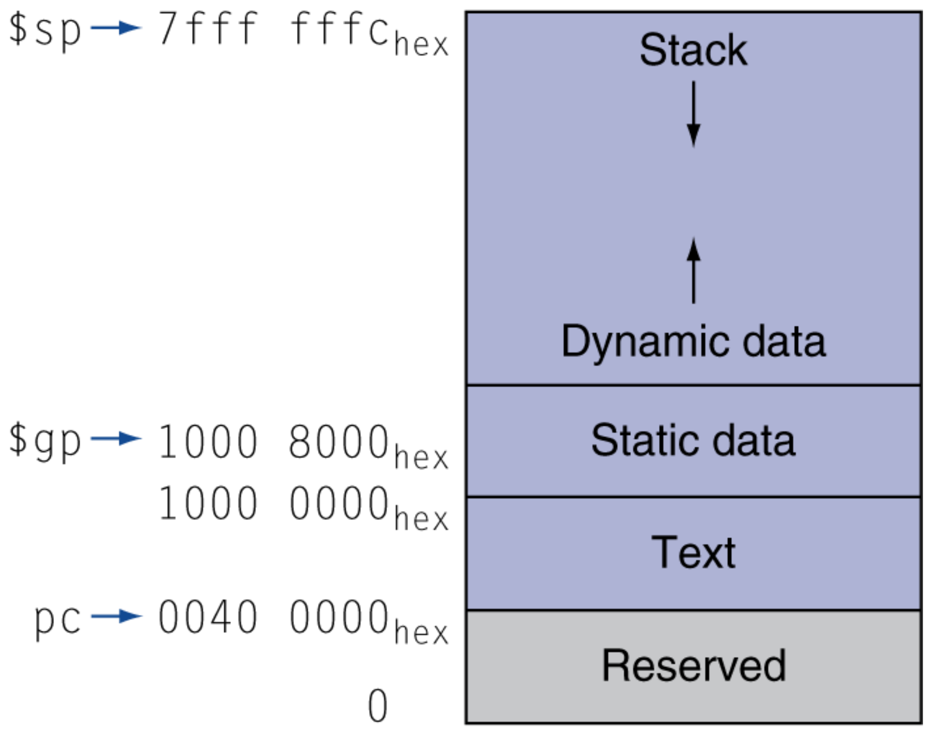
A. You should decrement the stack pointer ($sp) by 1

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 Store 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, -32  # Allocate space for stack frame
        sw      $ra, 28($sp)   # Stores (spills) $ra, return address
        sw      $s0, 24($sp)   # Stores (spills) s0, callee-saved reg
        …
        li      $s0, 25        # Set s0 to 25
        sw      $t3, 20($sp)   # Stores (spills) t3, caller-saved reg
        add     $a0, $t1, $t3
        jal     myFunction
        lw      $t3, 20($sp)   # Restores (fills) t3
        …
        lw      $s0,  24($sp)  # Restores (fills) s0, must restore
        lw      $ra,  28($sp)  # Restores (fills) $ra, return address
        addi    $sp, $sp, 32   # Restore the stack pointer
        jr      $ra            # Return
```

Complete example

```
foo:
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      $s0, 24($sp)
        …
        li      $s0, 25
        sw      $t3, 20($sp)
        add     $a0, $t1, $t3
        jal     myFunction
        lw      $t3, 20($sp)
        …
        lw      $s0,  24($sp)
        lw      $ra,  28($sp)
        addi    $sp, $sp, 32
        jr      $ra
```

Stack frame for foo (32 bytes in size)
Arguments are in $a0, …, $a3 and then on the stack at ($sp+32)+16, ($sp+32)+20, … for argument 5, 6, …

$sp + 28	Saved return address $ra
$sp + 24	Saved register $s0
$sp + 20	Saved register $t3
$sp + 16	Unused space to preserve 8-byte alignment
$sp + 12	Space for argument 4 (for use by myFunction)
$sp + 8	Space for argument 3 (for use by myFunction)
$sp + 4	Space for argument 2 (for use by myFunction)
$sp + 0	Space for argument 1 (for use by myFunction)