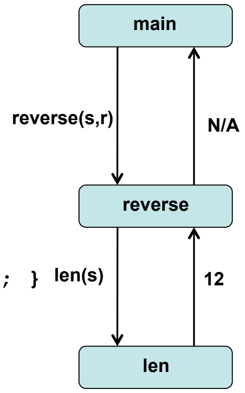
Procedures

```
int len(char *s) {
   for (int l=0; *s != ‘\0’; s++) l++;
   return l;
}
void reverse(char *s, char *r) {
   char *p, *t;
   int l = len(s);
   *(r+l) = ‘\0’;
   l--;
   for (p=s+l t=r; l>=0; l--) { *t++ = *p--; }
}
void main(int) {
   char *s = “Hello World!”;
   char r[100];
   reverse(s,r);
}
```



How can we do this with assembly?
- **Need a way to call / return procedures**
- **Need a way to pass arguments**
- **Need a way to return a value**

Procedure Call and Return

- Procedure call
  - Jump to the procedure
  - The return goes back to the point immediately after the call
  - Need to pass **“return address”** (instruction after call)
  - `jal Label`
    - $ra = PC+4 # set return address to next PC
    - PC = PC[31:28] | Label << 2 # jump to procedure
- Procedure return
  - Need return address (address of instruction after the `jal Label`)
  - Need to jump back to the return point
  - `jr $ra`
    - PC = $ra # jump back to return address

In Class Quick Example!

- Write a procedure “hello” that prints “Hello”
- Write a procedure “world” that prints “World”
- Write code to print “HelloWorld” using the procedures

In-Class Quick Example

```
        jal     world
        jal     hello
        jal     world
        li      $v0,10
        syscall
hello:  la      $a0,h_msg       # assume h_msg declared “Hello”
        li      $v0,4
        syscall
        jr      $ra
world:  la      $a0,w_msg       # assume w_msg declared “World”
        li      $v0,4
        syscall
        jr      $ra
```

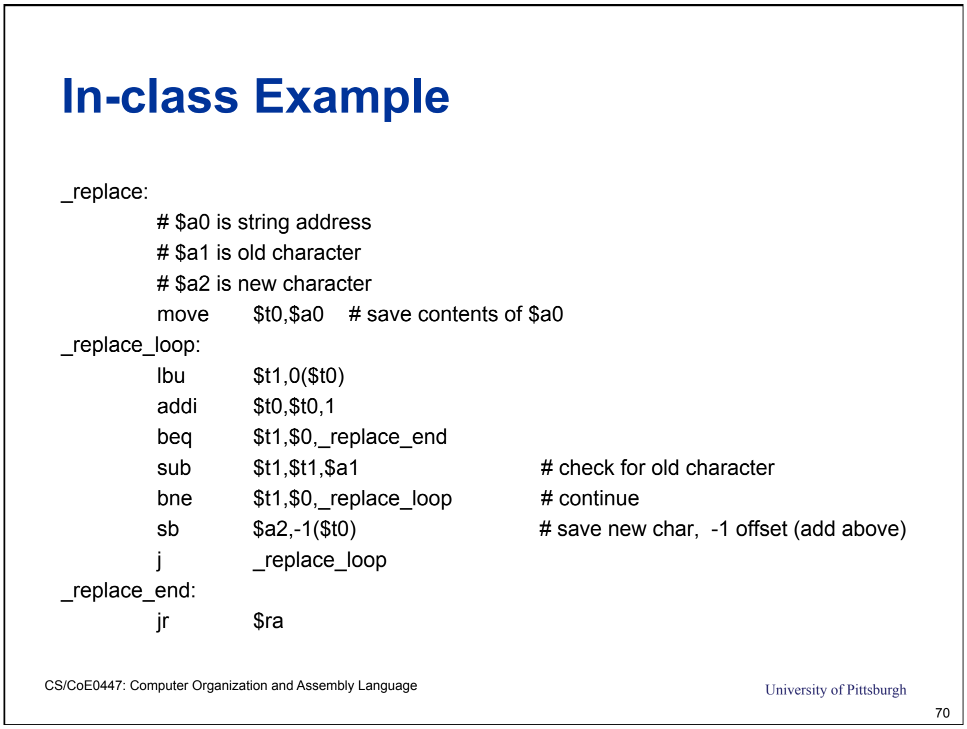
Arguments and Return Value

- Register conventions specified in PRM
  - $a0-$a3: four arguments for passing values to called procedure
  - $v0-$v1: two values returned from called procedure
  - $ra: return address register (set by call, used by return)
- Call chains
  - One procedure calls another, which calls another one
  - E.g., `main` → `reverse` → `len`
  - What happens to $ra??? (e.g., when reverse calls len)
- You must save $ra someplace!
  - Simple approach: A “free” register (can’t be used by caller)
  - **Leaf procedure**: Doesn’t make any calls. Doesn’t need to save $ra.

In-class Example

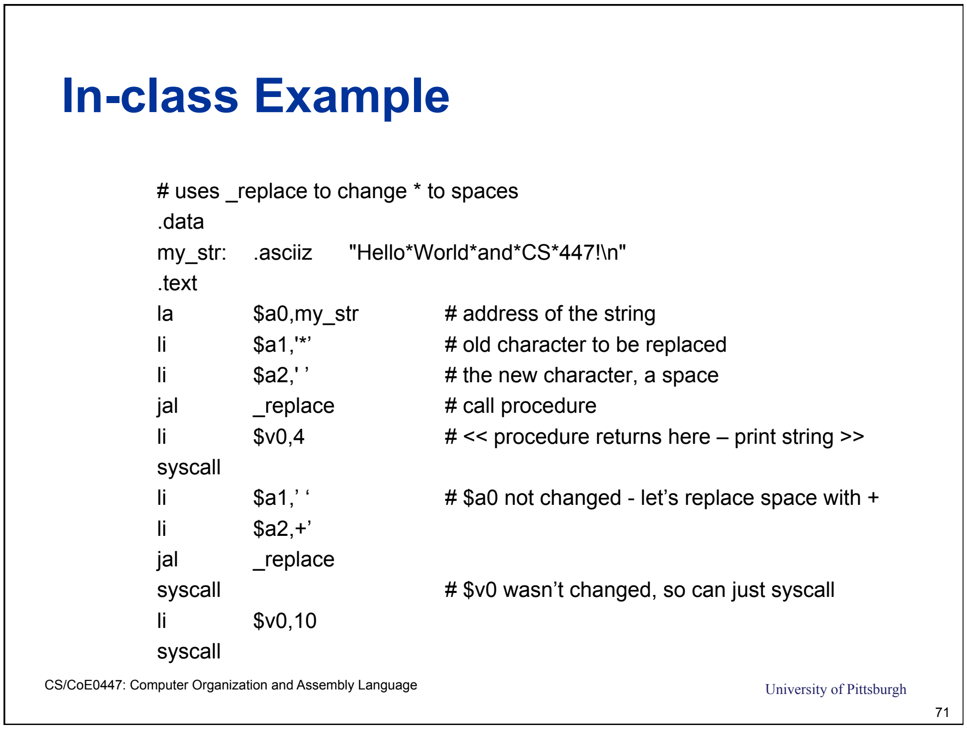
- Write a procedure that replaces a character with a new one in a string
- e.g., findReplace(char* string, char old, char new)

- Shows passing values with arguments
  - $a0 is the “string”
  - $a1 is the “old character”
  - $a2 is the “new character”



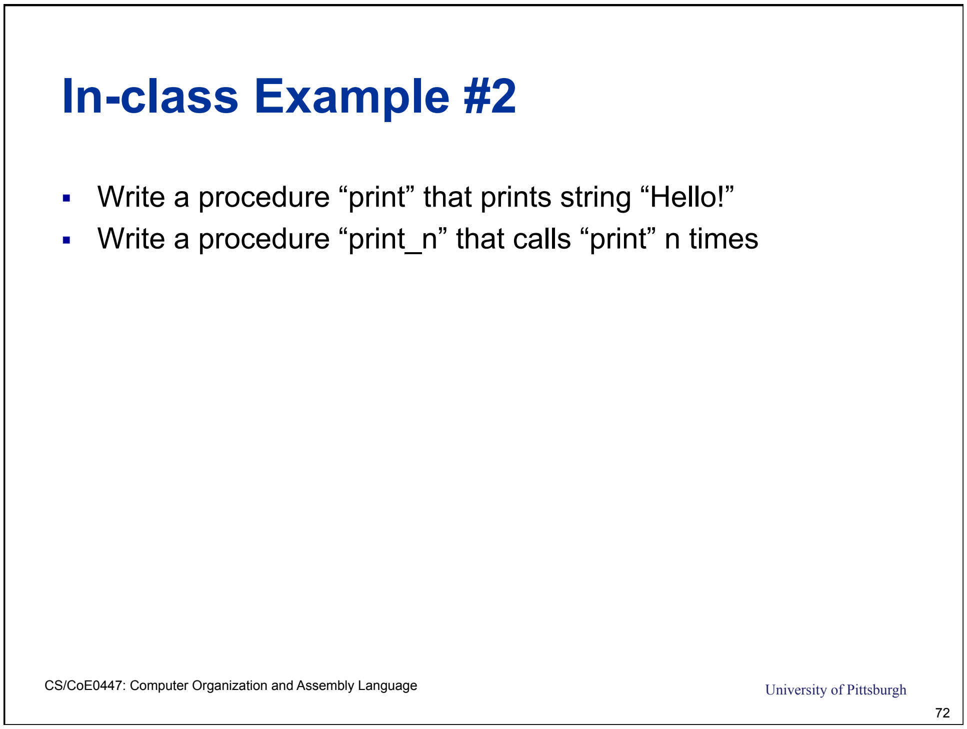
In-class Example

```
_replace:
        # $a0 is string address
        # $a1 is old character
        # $a2 is new character
        move    $t0,$a0    # save contents of $a0
_replace_loop:
        lbu     $t1,0($t0)
        addi    $t0,$t0,1
        beq     $t1,$0,_replace_end
        sub     $t1,$t1,$a1                 # check for old character
        bne     $t1,$0,_replace_loop        # continue
        sb      $a2,-1($t0)                 # save new char,  -1 offset (add above)
        j       _replace_loop
_replace_end:
        jr      $ra
```



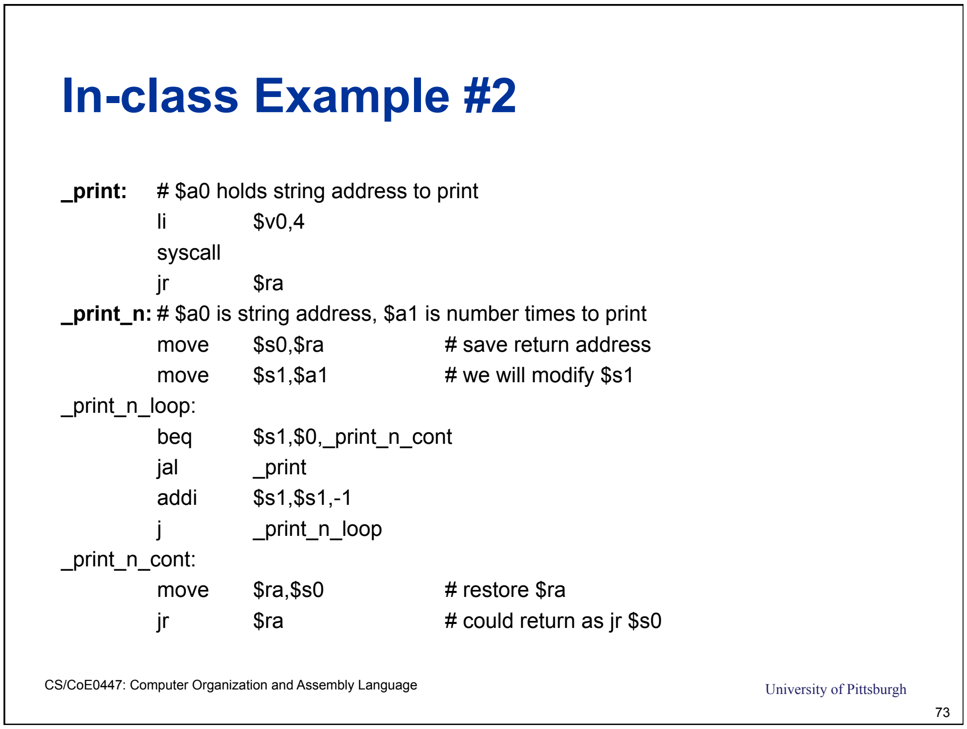
In-class Example

```
        # uses _replace to change * to spaces
        .data
        my_str:   .asciiz    "Hello*World*and*CS*447!\n"
        .text
        la        $a0,my_str          # address of the string
        li        $a1,'*’             # old character to be replaced
        li        $a2,' ’             # the new character, a space
        jal       _replace            # call procedure
        li        $v0,4               # << procedure returns here – print string >>
        syscall
        li        $a1,’ ‘             # $a0 not changed - let’s replace space with +
        li        $a2,+’
        jal       _replace
        syscall                       # $v0 wasn’t changed, so can just syscall
        li        $v0,10
        syscall
```



In-class Example #2

- Write a procedure “print” that prints string “Hello!”
- Write a procedure “print_n” that calls “print” n times



In-class Example #2

```
_print:   # $a0 holds string address to print
          li        $v0,4
          syscall
          jr        $ra
_print_n: # $a0 is string address, $a1 is number times to print
          move      $s0,$ra            # save return address
          move      $s1,$a1            # we will modify $s1
_print_n_loop:
          beq       $s1,$0,_print_n_cont
          jal       _print
          addi      $s1,$s1,-1
          j         _print_n_loop
_print_n_cont:
          move      $ra,$s0            # restore $ra
          jr        $ra                # could return as jr $s0
```

In-class Example #2

```
# calls print_n
.data
my_str:   .asciiz    "Hello!"
.text
la        $a0,my_str          # address of string to print
li        $a1,10              # number of times to print it
jal       _print_n            # call function print_n(string, n)
li        $v0,10              # exit service
syscall                       # terminate
```

In-class Example

- Change the “WorldHelloWorld” example to use one procedure that prints a string.

In-Class Example

```
         .data
w_msg:   .asciiz   “World”
h_msg:   .asciiz   “Hello”
         .data
         la        $a0,w_msg
         jal       print
         la        $a0,h_msg
         jal       print
         la        $a0,w_msg
         jal       print
         li        $v0,10
         syscall
print:   li        $v0,4
         syscall
         jr        $ra
```

Another example!

- Write two procedures
- Procedure #1: print(str): prints the string pointed to by str
- Procedure #2: hello(n): print “Hello World!” n times
  - Newline between each print
  - Shouldn’t print anything when n=0
  - What argument register to use?

See inclass5.asm

More Procedure Call/Return

- **Caller**: The procedure that calls another one
- **Callee**: The procedure that is called by the caller
- What if callee wants to use registers?
  - Caller is also using registers!!!
  - If callee wants to use same registers, it must save them
  - Consider what happened with $ra in a call chain
- Register usage conventions specified by PRM
  - $t0-$t9: Temp. registers; if caller wants them, must save before call
  - $s0-$s7: Saved registers; saved by callee prior to using them

Where to save?

- Need memory space to hold saved ("spilled") registers
  - Caller spills $t0-$t9 that be must saved to memory
  - Callee spills $s0-$s7 to memory, when these regs are used
  - Other registers (e.g., $v0, $v1 might also need to be saved)
  - Non-leaf caller saves $ra when making another call
- **Each procedure needs locations to save registers**
- In general, call-chain depth (number of called procs) is unknown, so we need to support undetermined length
- Suggestion: **Use a stack, located in memory.** Add "stack element" onto stack for each call. **The "stack element" has the locations to hold values**.

Program Stack

- **Program stack**: Memory locations used by running program
  - Has space for **saved registers**
  - Has space for **local variables**, when can't all fit in registers
    - E.g., local arrays are allocated on the stack
  - Has space for **return address**
- Each procedure allocates space for these items
  - So-called "**activation frame**" (a.k.a., "activation record")
  - Purpose of locations in activation frame are known
  - Location of activation frame isn't known until procedure call made
- **Prologue** (entry point into the procedure): Allocates an activation frame on the stack
- **Epilogue** (exit point from procedure): De-allocates the activation frame, does actual return

Procedure Structure and Stack

```
foo:
      # prologue - entry to function
      addi $sp, $sp, -8   # push, adjust size(AR)
      sw $s0,0($sp)       # save needed temp reg
      sw $ra,4($sp)       # save ra, non-leaf

        .... procedure body .....

foo_exit:
      # epilogue - exit from function
      lw $ra,4($sp)       # restore RA
      lw $s0,0($sp)       # restore $s0
      addi $sp,$sp,8      # pop AR

     jr $ra
```

(1) Before call

32		<$sp
28		
24		
20		
16		

(2) Prologue

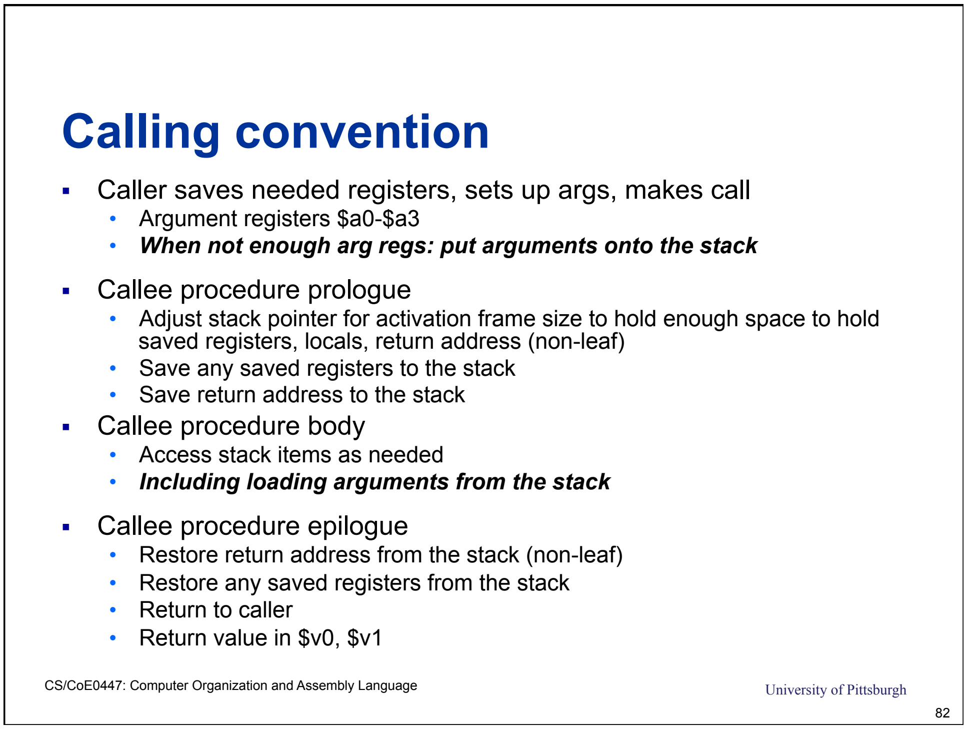
32		
28	$ra saved	
24	$s0 saved	<$sp
20		
16		

(3) Proc body

32		
28	$ra saved	
24	$s0 saved	<$sp
20		
16		

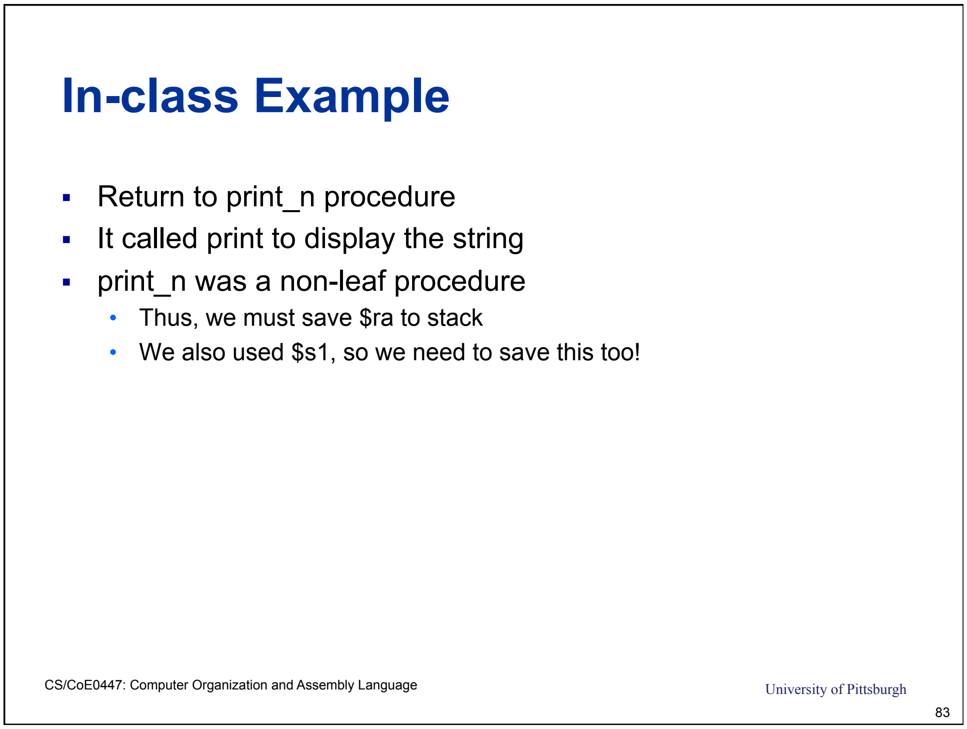
(4) Epilogue

32		<$sp
28		
24		
20		
16		



Calling convention

- Caller saves needed registers, sets up args, makes call
  - Argument registers $a0-$a3
  - ***When not enough arg regs: put arguments onto the stack***
- Callee procedure prologue
  - Adjust stack pointer for activation frame size to hold enough space to hold saved registers, locals, return address (non-leaf)
  - Save any saved registers to the stack
  - Save return address to the stack
- Callee procedure body
  - Access stack items as needed
  - ***Including loading arguments from the stack***
- Callee procedure epilogue
  - Restore return address from the stack (non-leaf)
  - Restore any saved registers from the stack
  - Return to caller
  - Return value in $v0, $v1



In-class Example

- Return to print_n procedure
- It called print to display the string
- print_n was a non-leaf procedure
  - Thus, we must save $ra to stack
  - We also used $s1, so we need to save this too!

In-class Example

```
_print_n:  # $a0 is string address, $a1 is number times to print
           addi    $sp,$sp,-8         # push frame: storing 8 bytes
           sw      $ra,0($sp)         # save return address -- not a leaf
           sw      $s1,4($sp)         # save $s1 by convention (caller, this proc, uses it)
           move    $s1,$a1            # use $s1 to hold $a1 across call
_print_n_loop:
           beq     $s1,$0,_print_n_cont
           jal     _print
           addi    $s1,$s1,-1
           j       _print_n_loop
_print_n_cont:
           lw      $ra,0($sp)         # restore $ra
           lw      $s1,4($sp)         # restore $s1
           addi    $sp,$sp,8          # pop stack frame
           jr      $ra                # return
```

Example: Factorial

```
/* factorial */
int fac(int f) {
  if (f == 1)   // end of recursion
    return 1;
  else          // go to bottom
    return (fac(f-1) * f);
}

int main(void) {
   a = fac(3);
   print(a);
}
```

Example: Factorial

fact(3)	returns 6
fact(3-1) * 3	returns 2 * 3
fact(2-1) * 2	returns 1 * 2
fact(1) * 1	returns 1 * 1

call factorial again, when not at end of recursion (f==1)
on each call, we need to pass a new argument to next one
on return, we do the actual computation and pass value back

need the return address & possibly temporary storage
set up a stack to make space

See factorial.asm