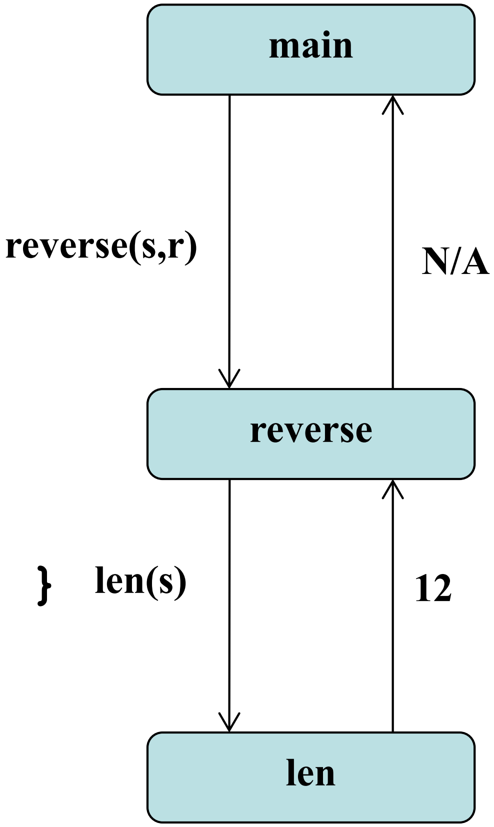
Procedures

```
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);
}
```



<u>How can we do this with assembly?</u>

- * 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”
- Call them both to print HelloWorld

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 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?

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

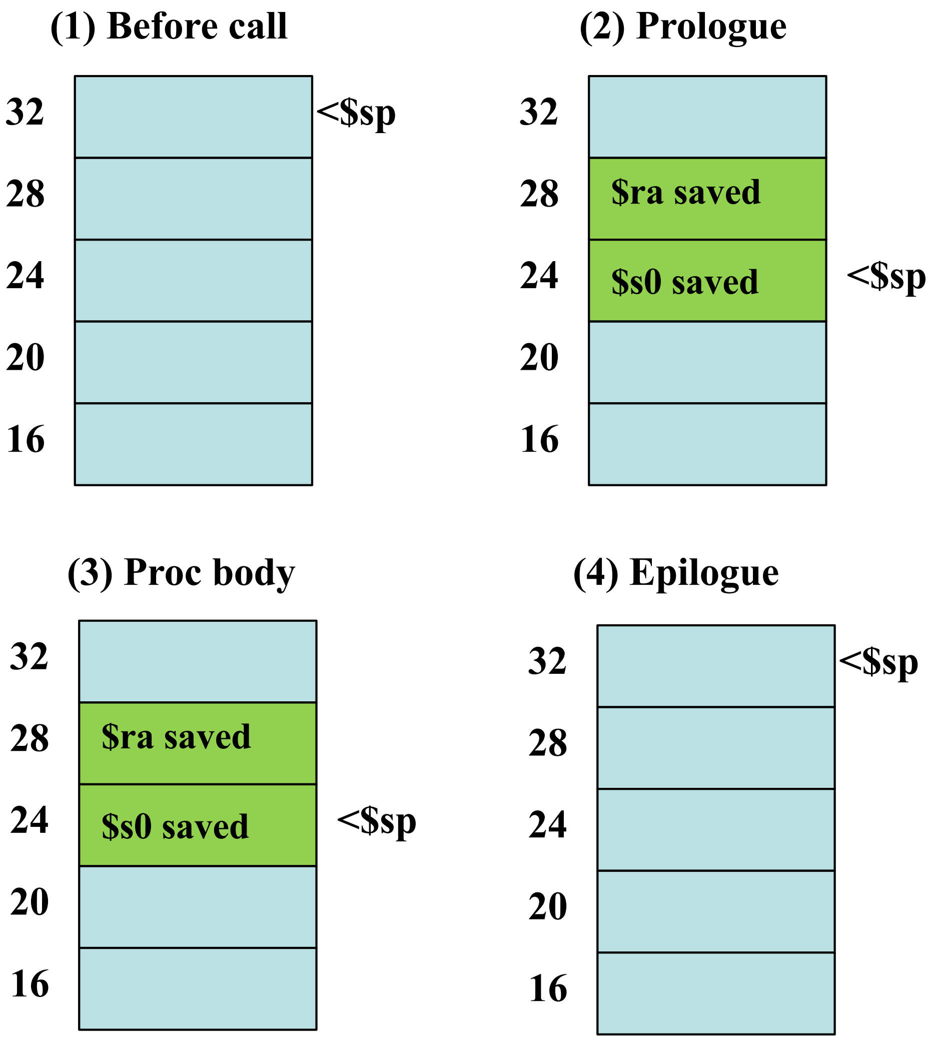
Procedure Structure and Stack

```
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

(2) Prologue

(3) Proc body

(4) Epilogue

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

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

```
        # procedure reverse($a0,$a1)
reverse:
        move    $t7,$ra        # save return address
        jal     len            # get length of source string
        blt     $v0,$0,rev_exit  # exit if empty string
        add     $t0,$a1,$v0    # null terminate target string
        sb      $0,0($t0)      # put null into end of string
        addi    $v0,$v0,-1     # decrement length (written /0)
        add     $t0,$a0,$v0    # $t0 holds p (source string)
        add     $t1,$a1,$0     # $t1 holds t (target string)
rev_loop:
        lbu     $t2,0($t0)     # get char from source string
        sb      $t2,0($t1)     # save char to target string
        addi    $t0,$t0,-1     # decrement source string ptr
        addi    $t1,$t1,1      # increment target string ptr
        addi    $v0,$v0,-1     # decrement length
        slt     $t2,$v0,$0     # is l < 0?
        beq     $t2,$0,rev_loop
rev_exit:
        move    $ra,$t7
        jr      $ra
        # procedure len($a0); returns string length in $v0
len:
        move    $t0,$a0               # copy start ptr
len_loop:
        lbu     $t1,0($t0)            # get char
        beq     $t1,$0,len_exit       # check for null
        addi    $t0,$t0,1             # go to next character
        j       len_loop              # continue loop
len_exit:
        sub     $v0,$t0,$a0           # diff of ptrs is length
        jr      $ra
```

```
.data
nl:     .asciiz         "\n"
s:      .asciiz         "Hello World!"
r:      .space          100
.align 2
p:      .word           0x0
t:      .word           0x0
l:      .word           0x0
.text
# make the call to reverse
la              $a0,s
la              $a1,r
jal             reverse
```

see mips12.asm for the full program