LAB 07: MIPS Functions and Stack Segment

Saleh AlSaleh
salehs@kfupm.edu.sa

King Fahd University of Petroleum and Minerals
College of Computing and Mathematics
Computer Engineering Department

COE301: Computer Architecture
Term 222

Agenda

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:
  - The **caller** must put the parameters (if there are) in a place where the **callee** function can access them.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:
  - The **caller** must put the parameters (if there are) in a place where the **callee** function can access them.
  - Transfer control to the **callee** function.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:
  - The **caller** must put the parameters (if there are) in a place where the **callee** function can access them.
  - Transfer control to the **callee** function.
  - Execute the **callee** function.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:
  - The **caller** must put the parameters (if there are) in a place where the **callee** function can access them.
  - Transfer control to the **callee** function.
  - Execute the **callee** function.
  - The **callee** function must put the results (if there are) in a place where the **caller** can access them.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:
  - The **caller** must put the parameters (if there are) in a place where the **callee** function can access them.
  - Transfer control to the **callee** function.
  - Execute the **callee** function.
  - The **callee** function must put the results (if there are) in a place where the **caller** can access them.
  - Return control to the **caller** (point of origin) next to where the call was made.

Caller vs. Callee

- The code/function that initiates the call to another function is known as **Caller**.
- The function that receives and executes the call is known as the **Callee**.
- To execute a function, the program must follow these steps:
  - The **caller** must put the parameters (if there are) in a place where the **callee** function can access them.
  - Transfer control to the **callee** function.
  - Execute the **callee** function.
  - The **callee** function must put the results (if there are) in a place where the **caller** can access them.
  - Return control to the **caller** (point of origin) next to where the call was made.

Functions: Definition, Execute(Call), Return Back

- Definition:
  - Define a label similar to if statements and loops.
  - Write the body of the function after the label.

Functions: Definition, Execute(Call), Return Back

- Definition:
  - Define a label similar to if statements and loops.
  - Write the body of the function after the label.
- Execution:
  - Prepare the arguments in $a0-$a3 registers.
  - Call the function using the jal instruction (e.g. jal function).

Functions: Definition, Execute(Call), Return Back

- Definition:
  - Define a label similar to if statements and loops.
  - Write the body of the function after the label.
- Execution:
  - Prepare the arguments in $a0-$a3 registers.
  - Call the function using the jal instruction (e.g. jal function).
- Return Back:
  - Prepare the results if any in $v0-$v1 registers.
  - Return to the caller using jr instruction (jr $ra)

Functions: Definition, Execute(Call), Return Back

- Definition:
  - Define a label similar to if statements and loops.
  - Write the body of the function after the label.
- Execution:
  - Prepare the arguments in $a0-$a3 registers.
  - Call the function using the jal instruction (e.g. jal function).
- Return Back:
  - Prepare the results if any in $v0-$v1 registers.
  - Return to the caller using jr instruction (jr $ra)

Function Example:

```
function: # function name
# function body
# return statement(essential!)
jr $ra
```

Registers Convention

Register Name	Register No.	Register Usage
$zero	$0	Always zero, forced by hardware
$at	$1	Assembler Temporary register, reserved for assembler use
$v0 - $v1	$2 - $3	Results of a function
$a0 - $a3	$4 - $7	Arguments of a function
$t0 - $t7	$8 - $15	Registers for storing temporary values
$s0 - $s7	$16 - $23	Registers that should be saved across function calls
$t8 - $t9	$24 - $25	Registers for storing more temporary values
$k0 - $k1	$26 - $27	Registers reserved for the OS kernel use
$gp	$28	Global Pointer register that points to global data
$sp	$29	Stack Pointer register that points to top of stack
$fp	$30	Frame Pointer register that points to stack frame
$ra	$31	Return Address register used to return from a function call

Stack Segment

- Stack Segment provides an area that can be allocated and freed by functions. The programmer has no control over where these segments are located in memory.

Stack Segment

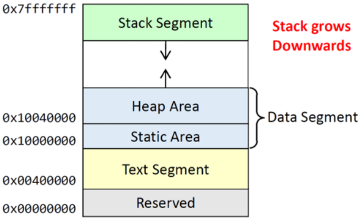
- Stack Segment provides an area that can be allocated and freed by functions. The programmer has no control over where these segments are located in memory.
- The stack segment can be used by functions for passing many parameters, for allocating space for local variables, and for saving and preserving registers across calls.

Stack Segment

- Stack Segment provides an area that can be allocated and freed by functions. The programmer has no control over where these segments are located in memory.
- The stack segment can be used by functions for passing many parameters, for allocating space for local variables, and for saving and preserving registers across calls.
- Without the stack segment in memory, it would be impossible to write recursive functions, or pure functions that have no side effects.

Stack Segment

- Stack Segment provides an area that can be allocated and freed by functions. The programmer has no control over where these segments are located in memory.
- The stack segment can be used by functions for passing many parameters, for allocating space for local variables, and for saving and preserving registers across calls.
- Without the stack segment in memory, it would be impossible to write recursive functions, or pure functions that have no side effects.



MIPS Memory Organization

Recursive Function Example

```c
int fact(int n){
  if (n < 2)
    return 1;
  else
    return n*fact(n-1);
}
```

Recursive factorial function

Recursive factorial function in MIPS Assembly

```
fact:
    bge $a0, 2, else        # branch if (n >= 2) to else
    li $v0, 1               # $v0 = 1
    jr $ra                  # return to caller
else:
    addi $sp, $sp, -8       # allocate 8 bytes in the stack
    sw $a0, 0($sp)          # save the argument n
    sw $ra, 4($sp)          # save the return address
    addi $a0, $a0, -1       # argument $a0 = n-1
    jal fact                # call fact(n-1)
    lw $a0, 0($sp)          # restore $a0 = n
    lw $ra, 4($sp)          # restore return address
    mul $v0, $a0, $v0       # $v0 = n * fact(n-1)
    addi $sp, $sp, 8        # free stack frame
    jr $ra                  # return to the caller
```

Live Examples

Task #1

Write a MIPS assembly program that that implements the read, reverse, and print functions used by f function in Figure 7.6 & Figure 7.7 in the PDF file. These functions should work with any size n (not only size 10). Then write a main function that calls function f.

Example function

```
void f() {
  int array[10];
  read(array, 10);
  reverse(array, 10);
  print(array, 10);
}
```

Stack Frame
saved $ra = 4 bytes
int array[10] (40 bytes)

f function in C

```
f:    addiu $sp, $sp, -44    # allocate stack frame = 44 bytes
      sw    $ra, 40($sp)     # save $ra on the stack
      move  $a0, $sp         # $a0 = address of array on the stack
      li    $a1, 10          # $a1 = 10
      jal   read             # call function read
      move  $a0, $sp         # $a0 = address of array on the stack
      li    $a1, 10          # $a1 = 10
      jal   reverse          # call function reverse
      move  $a0, $sp         # $a0 = address of array on the stack
      li    $a1, 10          # $a1 = 10
      jal   print            # call function print
      lw    $ra, 40($sp)     # load $ra from the stack
      addiu $sp, $sp, 44     # Free stack frame = 44 bytes
      jr    $ra              # return to caller
```

f function in MIPS Assembly

Task #1

Sample Run

```
Enter integer 1: 1
Enter integer 2: 2
Enter integer 3: 3
Enter integer 4: 4
Enter integer 5: 5
Enter integer 6: 6
Enter integer 7: 7
Enter integer 8: 8
Enter integer 9: 9
Enter integer 10: 10
Integer reversed = 10 9 8 7 6 5 4 3 2 1
```

Task #2

Write a MIPS assembly program that asks the user for an integer **n** he wishes to compute the Fibonacci number at that index. Calculate **fib(n)** based on the following code. Finally, print out the result.

```
int fib(int n) {
  if (n <= 1)
    return n;
  return fib(n-1)+fib(n-2);
}
```

Recursive Fibonacci function

Sample Run

```
Enter n: 7
fib(n) = 13
```