



# Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

## MIPS Assembly (Functions)

---

# Lab Schedule

## Activities

- **This Week**
  - Lab work time
  - MIPS functions
  - MIPS Random Number Generator

## Assignments Due

- **Lab 10**
  - **Due by Nov 17<sup>th</sup> 5:00am**
- **Lab 11**
  - **Due by Dec 1<sup>st</sup> 5:00am**
- **Lab 12**
  - **Due by Dec 8<sup>th</sup> 5:00am**
- **Final Exam**
  - **Tue Dec 9<sup>th</sup> 8-10am**

# MIPS Functions



# Function Requirements?

## ➔ **What happens when we call a function?**

1. Place function arguments in standard location where function can find them
2. Save current program location to return to later (the “Program Counter” register)
3. Jump to the function location
4. Function runs using provided arguments
5. Function produces output (return value) and saves it in standard location
6. Jump to original program location (return)
  1. *Technically, +1 instruction*

# Function Requirements

- **Can a function change local variables of its calling function?**
- No! The function operates in its own “bubble”
- **What happens if the function changes \$s0 which was also used by the calling function?**
- Problem! Your function has corrupted the calling function

# Functions in Assembly

In assembly, **you** must do all the background work for functions that the compiler did automatically in a higher level language

Functions still allow for **code re-use** (good!), but they're more complicated than in C or C++

# Registers

Name	Use
<b>\$zero</b>	Constant value: ZERO
<b>\$s0-\$s7</b>	Local variables (Convention: These are <i>saved</i> if a function needs to re-use them)
<b>\$t0-\$t9</b>	Temporary results (Convention: These are <i>not saved</i> if a function needs to re-use them)
<b>\$a0-\$a3</b>	Arguments to pass to function (max of 4)
<b>\$v0-\$v1</b>	Return value to obtain from function (max of 2)
<b>\$ra</b>	Return address of function
<b>\$sp</b>	Stack pointer (current top of stack)

**New!**

# More Jumps

- Jump and Link  
(side effect: `$ra` stores address of next instruction)

```
jal <destination>
```

Use this to *call* a function!

- Jump Register  
(destination address is stored in `<reg1>`)

```
jr <reg1>
```

Use this to *return from* a function!



# Task : Write Code

```
#include <stdio.h>

int function(int a);

int main()
{
    int x=5;
    int y;

    y = function(x);

    printf("y=%i\n", y);

    return 0;
}

int function(int a)
{
    return 3*a+5;
}
```

- Place arguments in  $\$a0-\$a3$
- Place return values in  $\$v0-\$v1$
- Return address saved automatically in  $\$ra$
- Ignore the stack for this example. (Thus, the function will destroy registers used by calling function)

```

# Simple routine to demo functions
# NOT using a stack in this example.
# Thus, the function does not preserve values
# of calling function!

# -----

        .text

        .globl main
main:
# Register assignments
# $s0 = x
# $s1 = y

# Initialize registers
lw      $s0, x      # Reg $s0 = x
lw      $s1, y      # Reg $s1 = y

# Call function
move    $a0, $s0    # Argument 1: x ($s0)
jal     fun         # Save current PC in $ra, and jump to fun
move    $s1, $v0    # Return value saved in $v0. This is y ($s1)

# Print msg1
li      $v0, 4      # print_string syscall code = 4
la      $a0, msg1
syscall

# Print result (y)
li      $v0, 1      # print_int syscall code = 1
move    $a0, $s1    # Load integer to print in $a0
syscall

# Print newline
li      $v0, 4      # print_string syscall code = 4
la      $a0, lf
syscall

# Exit
li      $v0, 10     # exit
syscall

```

```

# -----

# FUNCTION: int fun(int a)
# Arguments are stored in $a0
# Return value is stored in $v0
# Return address is stored in $ra (put there by jal instruction)
# Typical function operation is:

fun:    # Do the function math
        li $s0, 3
        mul $s1, $s0, $a0      # s1 = 3*$a0 (i.e. 3*a)
        addi $s1, $s1, 5      # 3*a+5

        # Save the return value in $v0
        move $v0, $s1

        # Return from function
        jr $ra                # Jump to addr stored in $ra

# -----

# Start .data segment (data!)
        .data
x:      .word 5
y:      .word 0
msg1:   .asciiz "y="
lf:     .asciiz "\n"

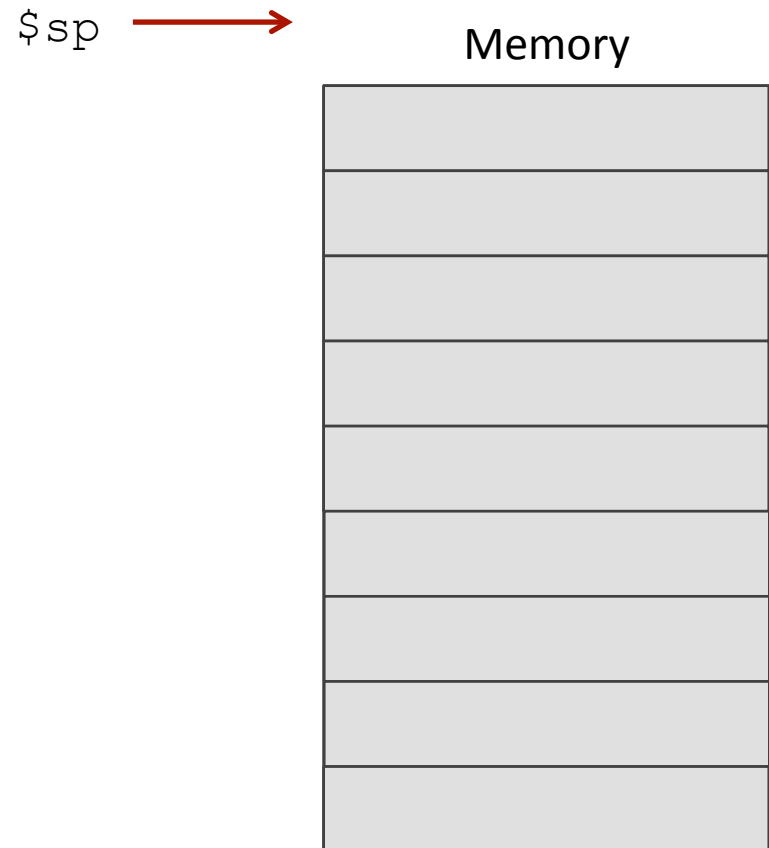
```

# Preserving Registers

- **What if we don't want to destroy registers used by the calling function?**
- Need to save those registers somewhere while our function runs (like memory!)
- A stack is a good structure for this

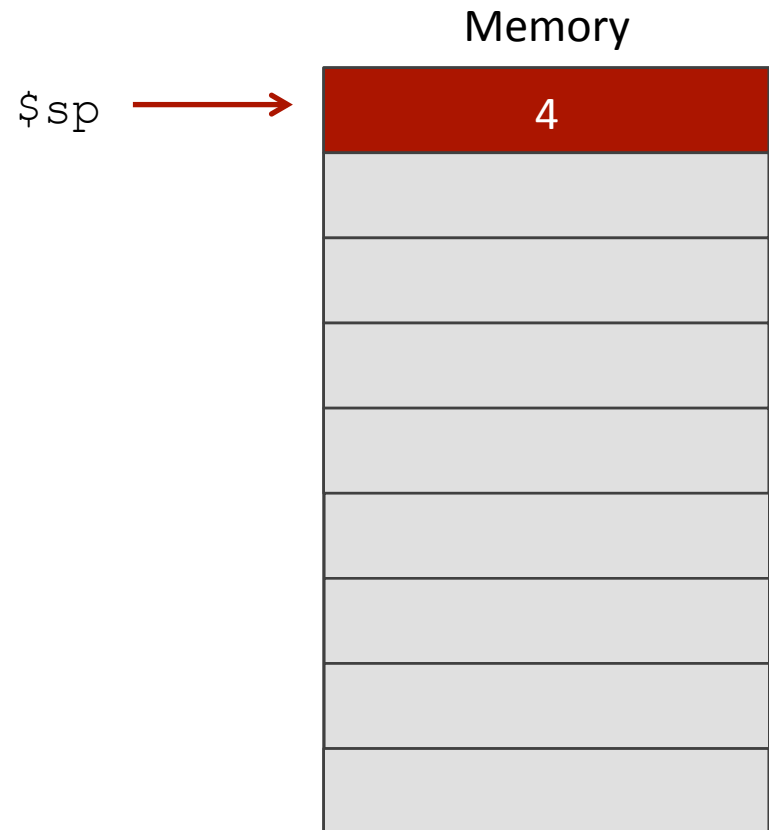
# The Stack

- Stack is a data structure stored in memory
- $\$sp$  (“Stack Pointer”) points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



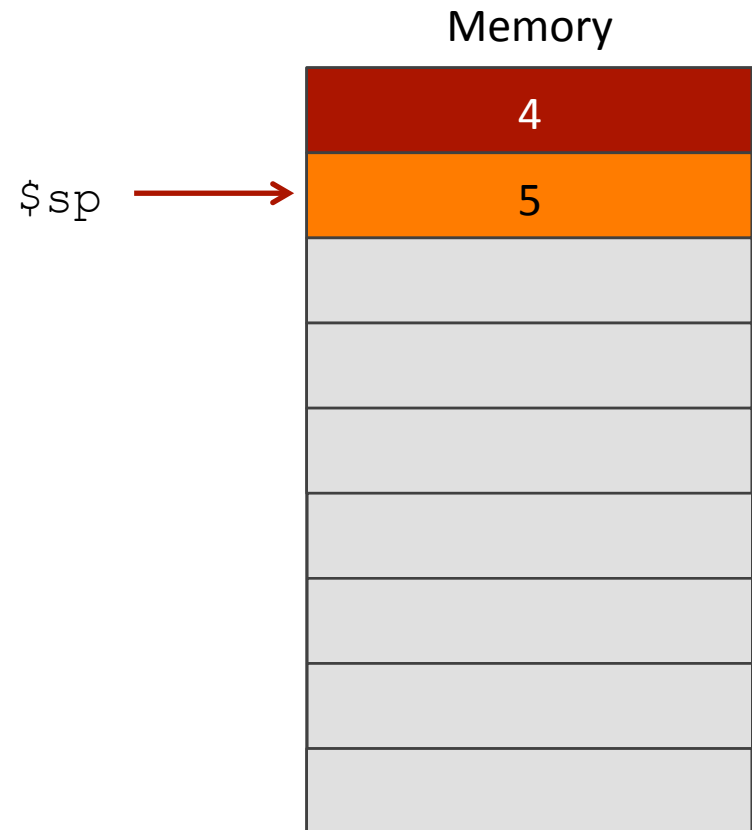
# The Stack

- Stack is a data structure stored in memory
- $\$sp$  (“Stack Pointer”) points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



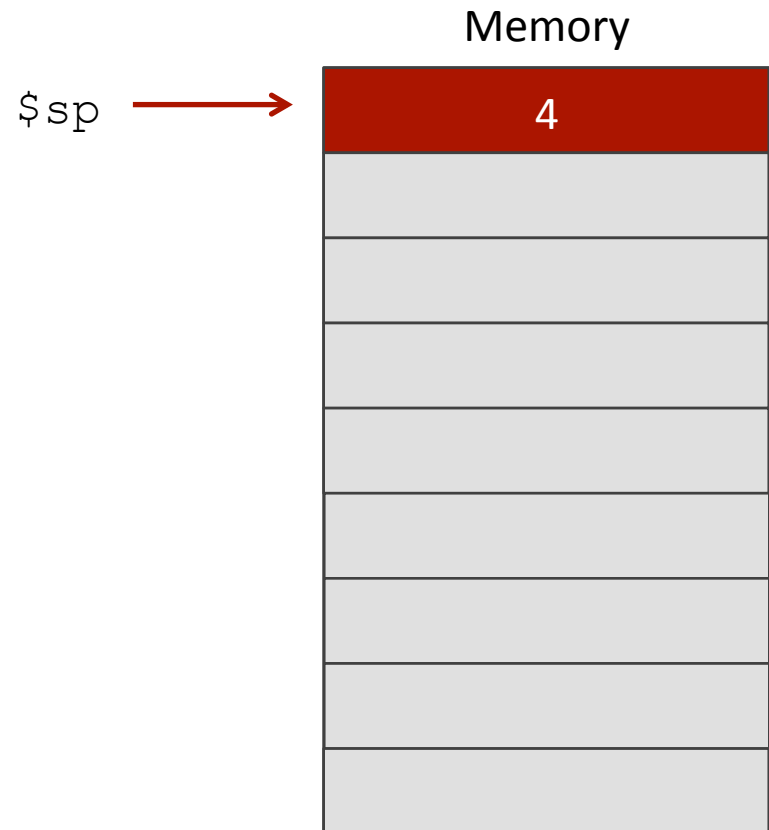
# The Stack

- Stack is a data structure stored in memory
- $\$sp$  (“Stack Pointer”) points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



# The Stack

- Stack is a data structure stored in memory
- $\$sp$  (“Stack Pointer”) points to top of stack
  - But stack grows **down** in memory!
- Example
  - Push 4 to stack
  - Push 5 to stack
  - Pop (5 from stack)
  - Pop (4 from stack)



# The Stack

- Stack is a data structure stored in memory
- $\$sp$  (“Stack Pointer”) points to top of stack
  - But stack grows **down** in memory!
- Example
  - Add 4 to stack
  - Add 5 to stack
  - Pop
  - Pop

$\$sp$  →

Memory





# The Stack

- **How would we modify previous solution to use a stack?**

```

# Simple routine to demo functions
# NOT using a stack in this example.
# Thus, the function does not preserve values
# of calling function!

# -----

        .text

        .globl main
main:
# Register assignments
# $s0 = x
# $s1 = y

# Initialize registers
lw      $s0, x      # Reg $s0 = x
lw      $s1, y      # Reg $s1 = y

# Call function
move    $a0, $s0    # Argument 1: x ($s0)
jal     fun         # Save current PC in $ra, and jump to fun
move    $s1, $v0    # Return value saved in $v0. This is y ($s1)

# Print msg1
li      $v0, 4      # print_string syscall code = 4
la      $a0, msg1
syscall

# Print result (y)
li      $v0, 1      # print_int syscall code = 1
move    $a0, $s1    # Load integer to print in $a0
syscall

# Print newline
li      $v0, 4      # print_string syscall code = 4
la      $a0, lf
syscall

# Exit
li      $v0, 10     # exit
syscall

```

```

# -----

# FUNCTION: int fun(int a)
# Arguments are stored in $a0
# Return value is stored in $v0
# Return address is stored in $ra (put there by jal instruction)
# Typical function operation is:

fun:    # This function overwrites $s0 and $s1
        # We should save those on the stack
        # This is PUSH'ing onto the stack
        addi $sp,$sp,-4      # Adjust stack pointer
        sw  $s0,0($sp)      # Save $s0
        addi $sp,$sp,-4      # Adjust stack pointer
        sw  $s1,0($sp)      # Save $s1

# Do the function math
li      $s0, 3
mul     $s1,$s0,$a0    # s1 = 3*$a0 (i.e. 3*a)
addi    $s1,$s1,5      # 3*a+5

# Save the return value in $v0
move    $v0,$s1

# Restore saved register values from stack in opposite order
# This is POP'ing from stack
lw  $s1,0($sp)      # Restore $s1
addi $sp,$sp,4      # Adjust stack pointer
lw  $s0,0($sp)      # Restore $s0
addi $sp,$sp,4      # Adjust stack pointer

# Return from function
jr      $ra            # Jump to addr stored in $ra

# -----

# Start .data segment (data!)
        .data
x:      .word 5
y:      .word 0
msg1:   .asciiz "y="
lf:     .asciiz "\n"

```

# Random Number Generator



# *In-Class Discussion*

# Square Root Calculation



# *In-Class Discussion*