# Unit 6: MIPS Function Calls

CSE 220: System Fundamental I

Stony Brook University

Joydeep Mitra

# Functions

- High-level languages use functions to enable code reusability and modularity
- Functions have input arguments and return values
- Functions (ideally) should not cause unintended side effects
- If function A calls function B, then A is the ***caller*** and B is ***callee***

# General Control Flow of MIPS Functions

- *Caller* places 4 arguments in registers **`$a0`**, **`$a1`**, **`$a2`**, and **`$a3`**
- *Caller* stores the **return address** in **`$ra`** and invokes *callee*
- *Callee* allocates space in memory, executes instructions in it, and places the return value in **`$v0`** and/or **`$v1`** before finishing
- *Callee* returns to the instruction in address **`$ra`** at the end of the function
- The programmer is responsible for managing control flow!
  - Fortunately, we have instructions and guidelines to help us

# MIPS Instructions to Call and Return

- MIPS uses instructions `jal` to call a function and `jr` to return from a function
- The `jal` instruction means jump to a label (denoting a function) after storing the *address of the next instruction* in **`$ra`**
- The `jr` instruction means jump to the instruction in register **`$ra`**

**High-Level Code**

```
int main() {
  simple();
  . . .
}
// void means the function returns no value
void simple() {
  return;
}
```

**MIPS Assembly Code**

```
0x00400200 main:    jal simple  # call function
0x00400204          . . .


0x00401020 simple:  jr $ra      # return
```

# Arguments and Return Values

- By convention,
  - the caller places arguments in registers **`$a0–$a3`**
  - the callee places return values in registers **`$v0–$v1`**
- Notice there are two registers to hold the return value. Why?
  - The result could be a 64-bit number
- What about more than 4 arguments? More on that soon!

**High-Level Code**

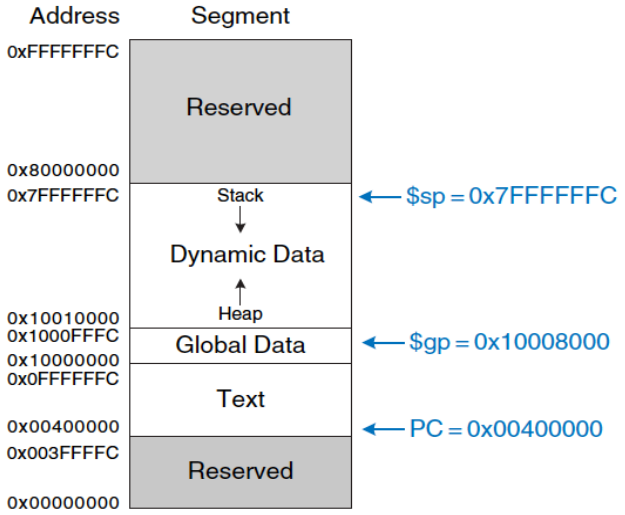
```
int main()
{
  int y;

  . . .

  y = diffofsums(2, 3, 4, 5);

  . . .
}

int diffofsums(int f, int g, int h, int i)
{
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

**MIPS Assembly Code**

```
# $s0 = y
main:
  . . .
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call function
  add  $s0, $v0, $0  # y = returned value
  . . .

# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

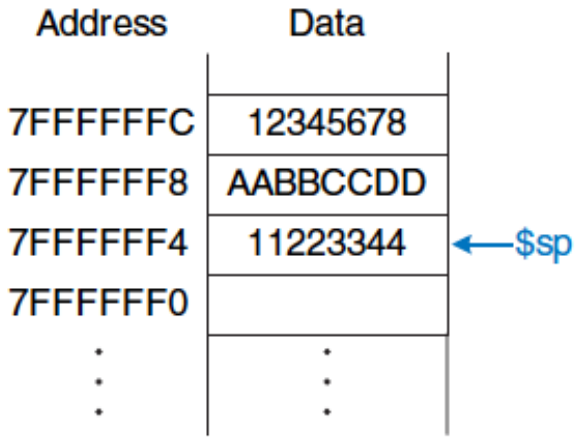
# How do Function Calls Work?

- Functions use the stack to save local variables they need
  - A stack is a *last-in-first-out* (LIFO) data structure
  - The MIPS stack starts at `$sp` (stack pointer) and grows **downward**, i.e., expands to lower memory adddresses
- Each function allocates stack space but must deallocate before returning




MIPS Memory


Stack with 1 element




Stack with 3 elements

# Purpose of The Stack

- Why do functions use stack?
- Recall that functions should not change any register except $v0 and $v1 (*side-effect free*)
- To prevent side-effect, functions allocate space on the stack to enable tracking intermediate computation

# Purpose of The Stack

- Let's examine the *diffofsums function:*
  - *diffofsums* uses registers that could be used by *main*
  - This could lead to error states; we want to avoid such situations

**High-Level Code**

```
int main()
{
  int y;

  . . .

  y = diffofsums(2, 3, 4, 5);

  . . .
}

int diffofsums(int f, int g, int h, int i)
{
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

**MIPS Assembly Code**

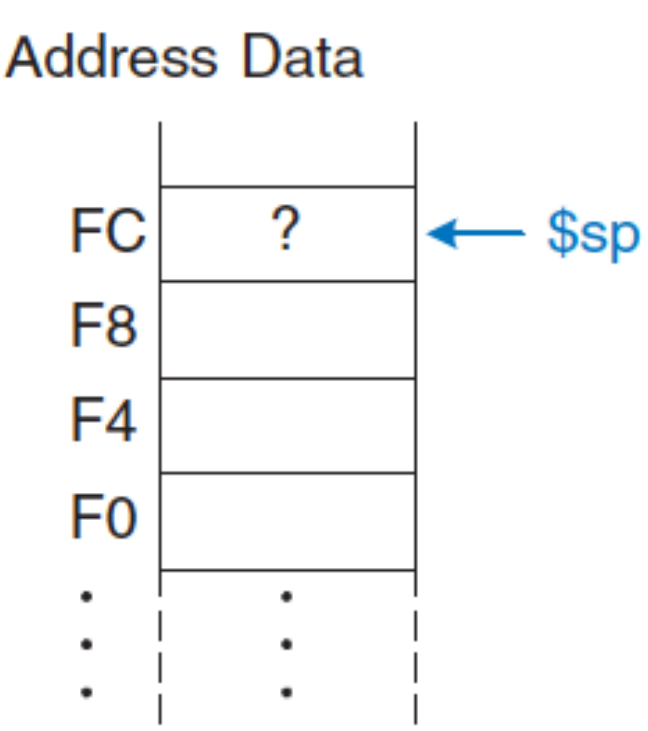
```
# $s0 = y
main:
  . . .
  addi $a0, $0, 2    # argument 0 = 2
  addi $a1, $0, 3    # argument 1 = 3
  addi $a2, $0, 4    # argument 2 = 4
  addi $a3, $0, 5    # argument 3 = 5
  jal  diffofsums    # call function
  add  $s0, $v0, $0  # y = returned value
  . . .

# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

# Managing The Stack

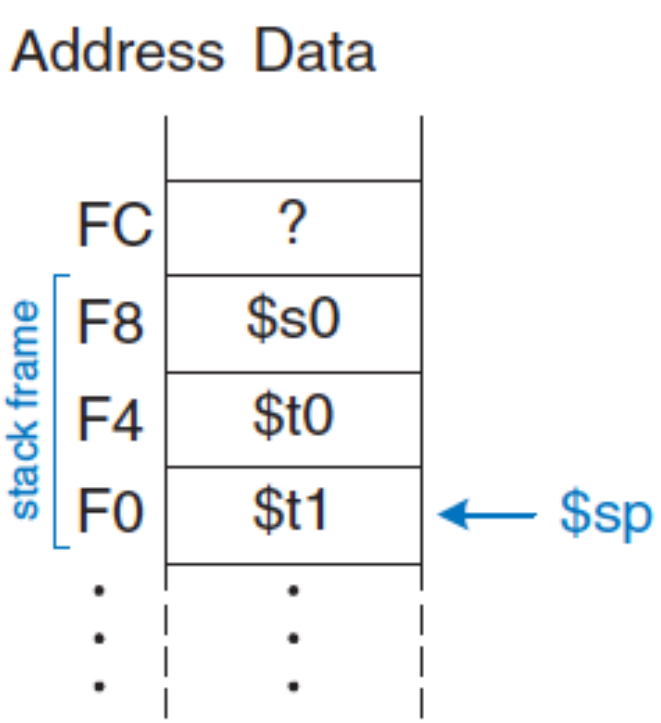
- To solve the problem, a function:
  - makes space on the stack for N register values
  - stores the values of N registers on the stack
  - executes the function using the registers
  - restores the original values of the registers from the stack
  - deallocates space on the stack before returning

```
# $s0 = result
diffofsums:
   addi $sp, $sp, -12 #  make space on stack to store three registers
   sw   $s0, 8($sp)   #  save $s0 on stack
   sw   $t0, 4($sp)   #  save $t0 on stack
   sw   $t1, 0($sp)   #  save $t1 on stack
   add  $t0, $a0, $a1 #  $t0 = f + g
   add  $t1, $a2, $a3 #  $t1 = h + i
   sub  $s0, $t0, $t1 #  result = (f + g) - (h + i)
   add  $v0, $s0, $0  #  put return value in $v0
   lw   $t1, 0($sp)   #  restore $t1 from stack
   lw   $t0, 4($sp)   #  restore $t0 from stack
   lw   $s0, 8($sp)   #  restore $s0 from stack
   addi $sp, $sp, 12  #  deallocate stack space
   jr   $ra           #  return to caller
```

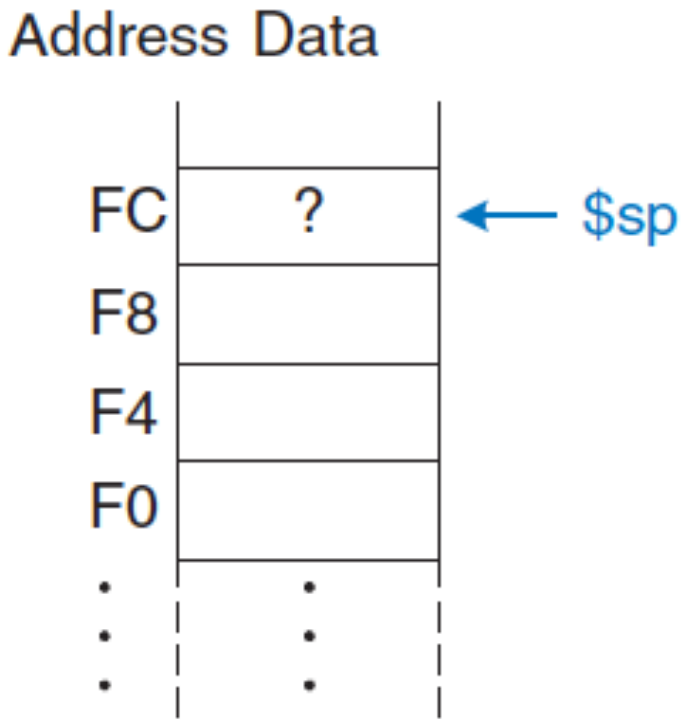
# Managing The Stack




The stack before *diffofsums*




The stack during *diffofsums*




The stack after *diffofsums*

# Register Conventions

- Do we have to save all registers in the stack frame?
  - No! that would be painful and wasteful!
- MIPS divides registers into *preserved ($s0-$s7)* and *non-preserved($t0-$t9)* registers
- The *callee* is expected to save and restore ONLY the *preserved* registers
- The *callee* may happily use *non-preserved* registers without saving
- The c*aller* is expected to save *non-preserved* registers before invoking the *callee*
- Hence, preserved registers are also called *callee-saved* and non-preserved registers are called *caller-saved*

# Improved Example

- Let's use the register conventions for our *diffofsums* example

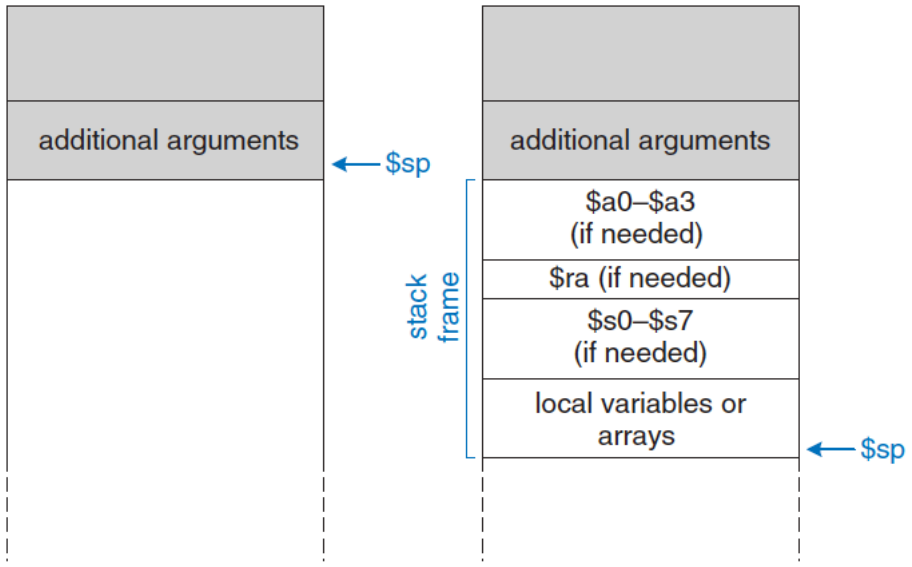
```
# $s0 = result
diffofsums
   addi  $sp, $sp, -4      # make space on stack to store one register
   sw    $s0, 0($sp)       # save $s0 on stack
   add   $t0, $a0, $a1     # $t0 = f + g
   add   $t1, $a2, $a3     # $t1 = h + i
   sub   $s0, $t0, $t1     # result = (f + g) - (h + i)
   add   $v0, $s0, $0      # put return value in $v0
   lw    $s0, 0($sp)       # restore $s0 from stack
   addi  $sp, $sp, 4       # deallocate stack space
   jr    $ra               # return to caller
```

# Summary of Register Classification

| Preserved | Nonpreserved |
|---|---|
| Saved registers: `$s0–$s7` | Temporary registers: `$t0–$t9` |
| Return address: `$ra` | Argument registers: `$a0–$a3` |
| Stack pointer: `$sp` | Return value registers: `$v0–$v1` |
| Stack above the stack pointer | Stack below the stack pointer |

# More Than Four Arguments

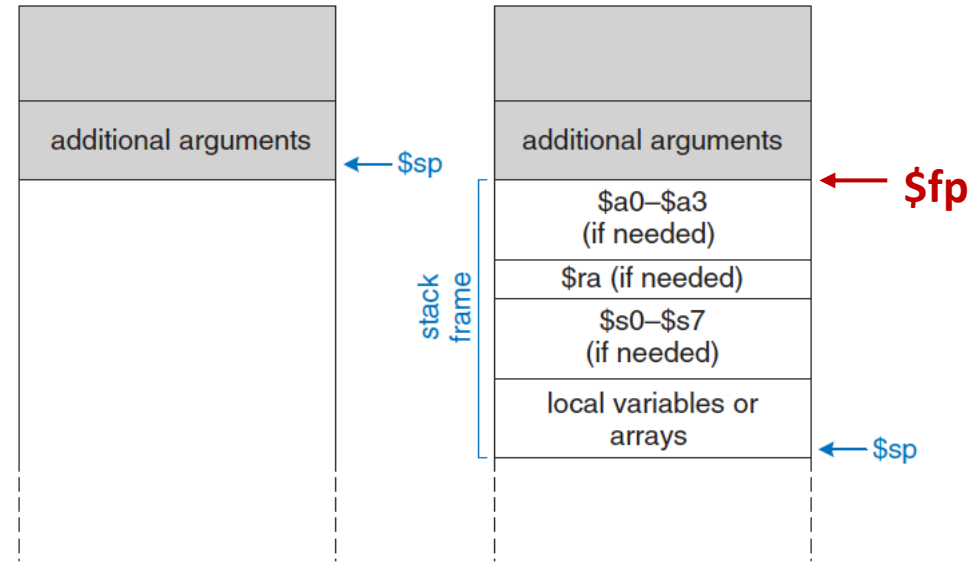
- For functions with more than 4 arguments, the *caller* pushes the additional arguments into its *own stack frame* just above `$sp`
- The *callee* accesses the additional arguments from the caller's stack frame. How?
  - By adjusting `$sp`
- Accessing stack frame of caller from callee may introduce side effect
  - Its an acceptable tradeoff




Stack Before Call  Stack After Call

# Frame Pointer

- The callee function has to adjust **$sp** to store local variables or preserved registers
- A moving **$sp** makes it inconvenient to access additional arguments
- Hence, at the start of the function call we move **$sp** to a preserved register ($fp); also called frame pointer
- **$fp** allows the callee to change **$sp** without worrying about accessing additional arguments




Stack Before Call

Stack After Call

$sp changes; $fp is constant

# *diffofsums* With Five Arguments

```
.text
main:
    addi $a0, $0, 2     # argument 0 = 2
    addi $a1, $0, 3     # argument 1 = 3
    addi $a2, $0, 4     # argument 2 = 4
    addi $a3, $0, 5     # argument 3 = 5
    addi $t0, $0, 10    # additional argument = 10
    addi $sp, $sp, -4   # make space on main's stack frame
    sw $t0, 0($sp)      # store additional argument
    jal diffofsums
    move $s0, $v0
    li $v0, 10
    syscall

diffofsums:
    move $fp, $sp       # save current stack pointer
    addi $sp, $sp, -4   # make space on diffofsums stack frame
    sw $s0, 0($sp)      # save one preserved register
    add $t0, $a0, $a1   # $t0 = arg0 + arg1
    lw $t1, 0($fp)      # $t1 = additional arg from main's stack frame
    add $t0, $t0, $t1   # $t0 = $t0 + $t1
    add $t2, $a2, $a3   # $t2 = arg2 + arg3
    sub $s0, $t0, $t2   # $s0 = $t0 - $t2
    add $v0, $0, $s0    # $v0 is return value
    lw $s0, 0($sp)      # restore preserved register
    addi $sp, $sp, 4    # deallocate stack
    jr $ra              # return to caller
```

# Nested Function Calls

- When a function *f1* invokes another function *f2*, *f1's* return address must be preserved in *f1's* stack frame
- E.g., *diffsums* calls a function *is_pos* to find out if `$a1 < 0`

```
diffofsums:
    move $fp, $sp              # save current stack pointer
    addi $sp, $sp, -12         # make space on diffofsums stack frame
    sw $a0, 0($sp)             # save arg0 on diffofsums stack frame
    sw $ra, 4($sp)             # save $ra on diffofsums stack frame
    sw $s0, 8($sp)             # save one preserved register
    move $a0, $a1              # prepare arg0 for is_pos function
    jal is_pos                 # call is_pos
    move $s0, $v0              # $s0 = return val from is_pos
    beq $s0, $0, negFound      # jump to negFound if $s0 = 0
    lw $a0, 0($sp)             # restore arg0 to diffsums from stack
    add $t0, $a0, $a1          # $t0 = arg0 + arg1
    lw $t1, 0($fp)             # $t1 = additional arg from main's stack frame
    add $t0, $t0, $t1          # $t0 = $t0 + $t1
    add $t2, $a2, $a3          # $t2 = arg2 + arg3
    sub $s0, $t0, $t2          # $s0 = $t0 - $t2
    add $v0, $0, $s0           # $v0 is return value
    j retn                     # jump to returning value
```

# Nested Function Calls (cont.)

```
    negFound:
        addi $v0, $0, -1        # set return value to -1
    retn:
        lw $s0, 0($sp)          # restore preserved register
        lw $ra, 4($sp)          # restore return address of diffofsums
        addi $sp, $sp, 12       # deallocate stack
        jr $ra                  # return to caller

is_pos:
    blt $a0, $0, num_neg        # if arg0 < 0 then jump to num_neg
    addi $v0, $0, 1             # set return value to 1
    jr $ra                      # return to caller
    num_neg:
        addi $v0, $0, 0         # set return value to 0
        jr $ra                  # return to caller
```