

# MIPS Functions and the Runtime Stack

COE 301

Computer Organization

Prof. Muhamed Mudawar

College of Computer Sciences and Engineering  
King Fahd University of Petroleum and Minerals

# Presentation Outline

- ❖ **Functions**
- ❖ **Function Call and Return**
- ❖ The Stack Segment
- ❖ Preserving Registers
- ❖ Allocating a Local Array on the Stack
- ❖ Examples: Bubble Sort and Recursion

# Functions

- ❖ A function (or a procedure) is a block of instructions that can be called at several different points in the program
  - ✧ Allows the programmer to focus on just one task at a time
  - ✧ Allows code to be reused
- ❖ The function that initiates the call is known as the **caller**
- ❖ The function that receives the call is known as the **callee**
- ❖ When the callee finishes execution, control is transferred back to the caller function.
- ❖ A function can receive **parameters** and return **results**
- ❖ The function parameters and results act as an interface between a function and the rest of the program

# Function Call and Return

- ❖ To execution a function, the **caller** does the following:
  - ✧ Puts the parameters in a place that can be accessed by the callee
  - ✧ Transfer control to the callee function
- ❖ To return from a function, the **callee** does the following:
  - ✧ Puts the results in a place that can be accessed by the caller
  - ✧ Return control to the caller, next to where the function call was made
- ❖ Registers are the fastest place to pass parameters and return results. The MIPS architecture uses the following:
  - ✧ **\$a0-\$a3**: four argument registers in which to pass parameters
  - ✧ **\$v0-\$v1**: two value registers in which to pass function results
  - ✧ **\$ra**: return address register to return back to the caller

# Function Call and Return Instructions

- ❖ **JAL (Jump-and-Link)** is used to call a function
  - ✧ Save return address in **\$31 = PC+4** and jump to function
  - ✧ Register **\$31 (\$ra)** is used by **JAL** as the **return address**
- ❖ **JR (Jump Register)** is used to return from a function
  - ✧ Jump to instruction whose address is in register Rs (PC = Rs)
- ❖ **JALR (Jump-and-Link Register)**
  - ✧ Save return address in Rd = PC+4, and
  - ✧ Call function whose address is in register Rs (PC = Rs)
  - ✧ Used to call functions whose addresses are known at runtime

Instruction	Meaning	Format						
<b>jal label</b>	<b>\$31 = PC+4, j Label</b>	<b>Op=3</b>	<b>26-bit address</b>					
<b>jr Rs</b>	<b>PC = Rs</b>	<b>Op=0</b>	<b>Rs</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>8</b>
<b>jalr Rd, Rs</b>	<b>Rd = PC+4, PC = Rs</b>	<b>Op=0</b>	<b>Rs</b>	<b>0</b>	<b>Rd</b>	<b>0</b>	<b>0</b>	<b>9</b>

# Example

- ❖ Consider the following **swap** function (written in C)
- ❖ Translate this function to MIPS assembly language

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

## Parameters:

**\$a0** = Address of **v[]**

**\$a1** = **k**, and

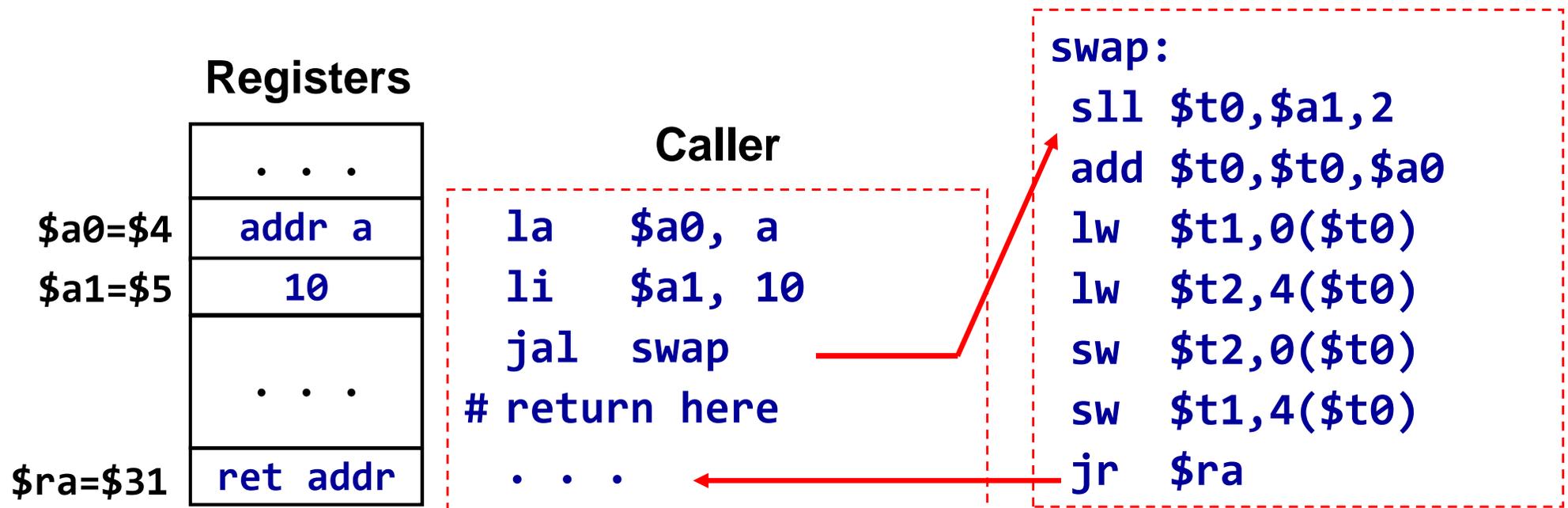
Return address is in **\$ra**

## swap:

```
sll $t0,$a1,2      # $t0=k*4
add $t0,$t0,$a0    # $t0=v+k*4
lw  $t1,0($t0)     # $t1=v[k]
lw  $t2,4($t0)     # $t2=v[k+1]
sw  $t2,0($t0)     # v[k]=$t2
sw  $t1,4($t0)     # v[k+1]=$t1
jr  $ra           # return
```

# Call / Return Sequence

- ❖ Suppose we call function swap as: **swap(a, 10)**
  - ✧ Pass **address** of array **a** and **10** as arguments
  - ✧ Call the function swap saving **return address** in **\$31 = \$ra**
  - ✧ Execute function swap
  - ✧ Return control to the point of origin (return address)



# Details of JAL and JR

Address	Instructions	Assembly Language
00400020	lui \$1, 0x1001	la \$a0, a
00400024	ori \$4, \$1, 0	
00400028	ori \$5, \$0, 10	ori \$a1,\$0,10
0040002C	jal 0x10000f	jal swap
00400030	...	# return here
0040003C	sll \$8, \$5, 2	swap: sll \$t0, \$a1, 2
00400040	add \$8, \$8, \$4	add \$t0, \$t0, \$a0
00400044	lw \$9, 0(\$8)	lw \$t1, 0(\$t0)
00400048	lw \$10, 4(\$8)	lw \$t2, 4(\$t0)
0040004C	sw \$10, 0(\$8)	sw \$t2, 0(\$t0)
00400050	sw \$9, 4(\$8)	sw \$t1, 4(\$t0)
00400054	jr \$31	jr \$ra

## Pseudo-Direct Addressing

$$PC = imm26 \ll 2$$

$$0x10000f \ll 2$$

$$= 0x0040003C$$

\$31

0x00400030

Register \$31  
is the return  
address register

# Second Example

- ❖ Function `tolower` converts a capital letter to lowercase
- ❖ If parameter `ch` is not a capital letter then return `ch`

```
char tolower(char ch) {  
    if (ch>='A' && ch<='Z')  
        return (ch + 'a' - 'A');  
    else  
        return ch;  
}
```

`tolower:`

```
    blt    $a0, 'A', else
```

```
    bgt    $a0, 'Z', else
```

```
    addi   $v0, $a0, 32
```

```
    jr     $ra
```

`else:`

```
    move   $v0, $a0
```

```
    jr     $ra
```

```
# $a0 = parameter ch
```

```
# branch if $a0 < 'A'
```

```
# branch if $a0 > 'Z'
```

```
# 'a' - 'A' == 32
```

```
# return to caller
```

```
# $v0 = ch
```

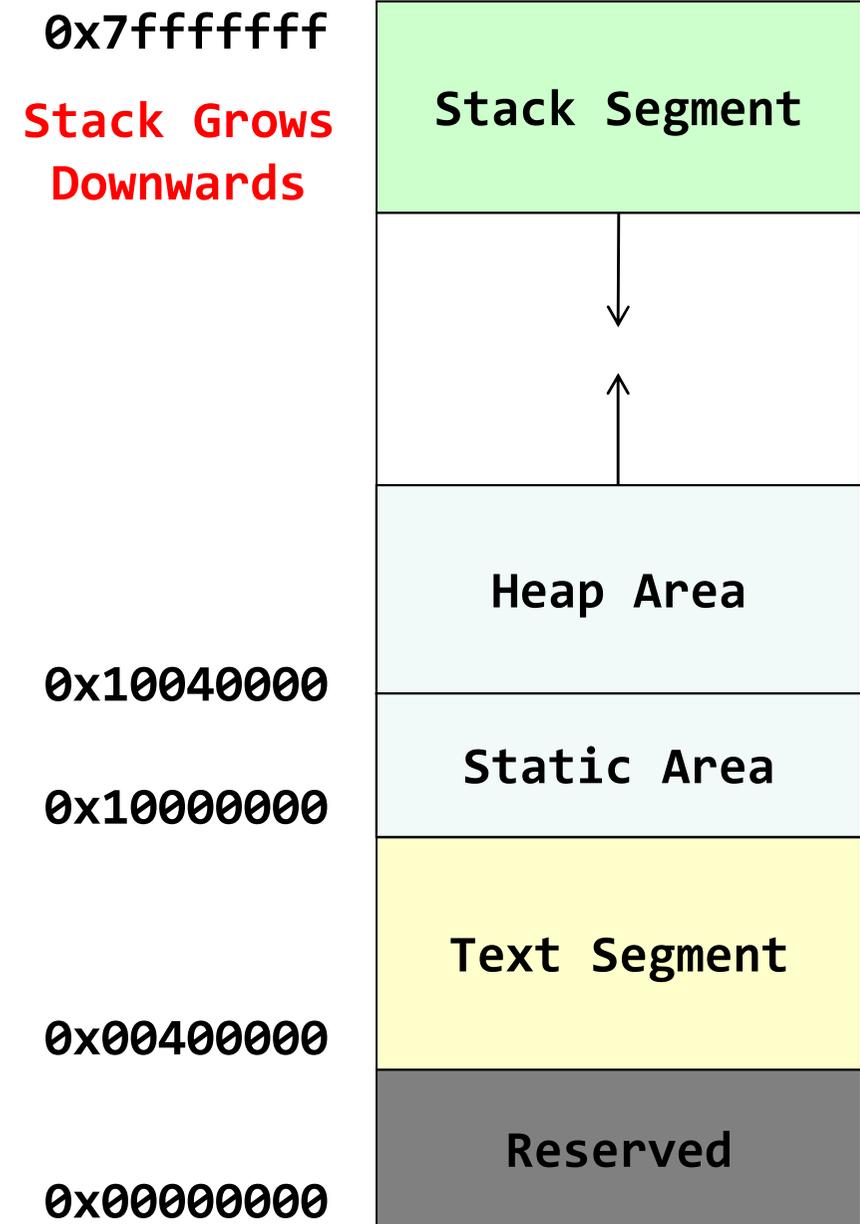
```
# return to caller
```

# Next ...

- ❖ Functions
- ❖ Function Call and Return
- ❖ **The Stack Segment**
- ❖ **Preserving Registers**
- ❖ Allocating a Local Array on the Stack
- ❖ Examples: Bubble Sort and Recursion

# The Stack Segment

- ❖ Every program has 3 segments when loaded into memory:
  - ✧ **Text segment:** stores machine instructions
  - ✧ **Data segment:** area used for static and dynamic variables
  - ✧ **Stack segment:** area that can be allocated and freed by functions
- ❖ The program uses only logical (virtual) addresses
- ❖ The actual (physical) addresses are managed by the OS

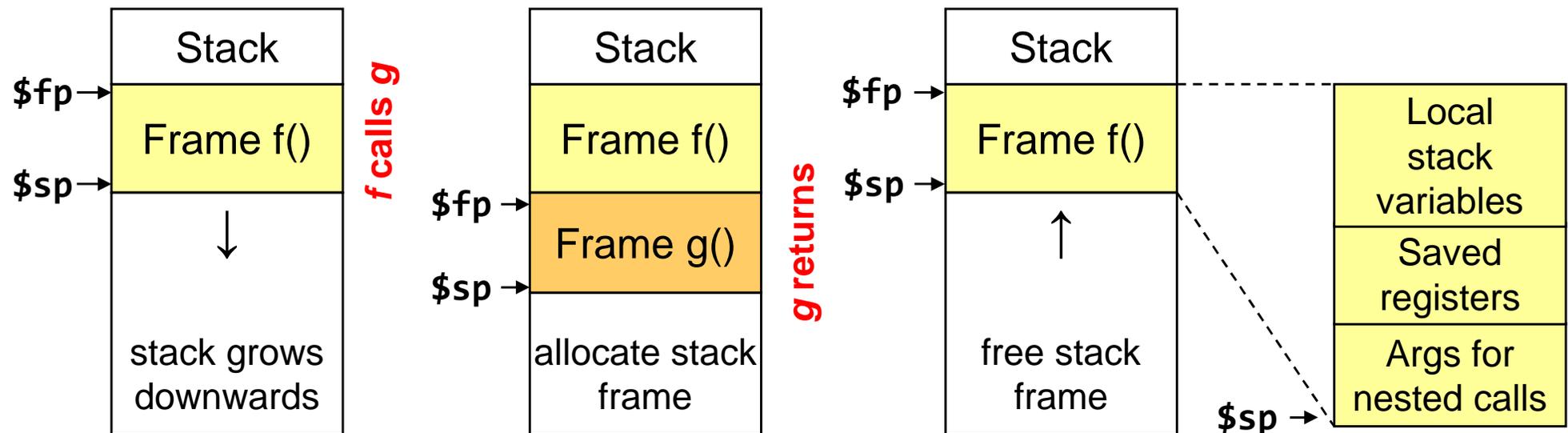


# The Stack Segment (cont'd)

- ❖ The stack segment is used by functions for:
  - ✧ Passing parameters that cannot fit in registers
  - ✧ Allocating space for local variables
  - ✧ Saving registers across function calls
  - ✧ Implement recursive functions
- ❖ The stack segment is implemented via software:
  - ✧ The **Stack Pointer \$sp = \$29** (points to the top of stack)
  - ✧ The **Frame Pointer \$fp = \$30** (points to a stack frame)
- ❖ The stack pointer **\$sp** is initialized to the base address of the stack segment, just before a program starts execution
- ❖ The MARS tool initializes register **\$sp** to **0x7ffeffc**

# Stack Frame

- ❖ **Stack frame** is an area of the stack containing ...
  - ❖ Saved arguments, registers, local arrays and variables (if any)
- ❖ Called also the **activation frame**
- ❖ Frames are pushed and popped by adjusting ...
  - ❖ Stack pointer  $\$sp = \$29$  (and sometimes frame pointer  $\$fp = \$30$ )
  - ❖ Decrement  $\$sp$  to allocate stack frame, and increment to free



# Leaf Function

- ❖ A leaf function does its work without calling any function
- ❖ Example of leaf functions are: **swap** and **tolower**
- ❖ A leaf function can freely modify some registers:
  - ✧ Argument registers: **\$a0 - \$a3**
  - ✧ Result registers: **\$v0 - \$v1**
  - ✧ Temporary registers: **\$t0 - \$t9**
  - ✧ These registers can be modified without saving their old values
- ❖ A leaf function does not need a stack frame if ...
  - ✧ Its variables can fit in temporary registers
- ❖ A leaf function allocates a stack frame only if ...
  - ✧ It requires additional space for its local variables

# Non-Leaf Function

- ❖ A non-leaf function is a function that calls other functions
- ❖ A non-leaf function must allocate a stack frame
- ❖ Stack frame size is computed by the programmer (compiler)
- ❖ To allocate a stack frame of **N** bytes ...
  - ✧ Decrement **\$sp** by **N** bytes:  $\text{\$sp} = \text{\$sp} - N$
  - ✧ **N** must be multiple of **4** bytes to have registers aligned in memory
  - ✧ In our examples, only register **\$sp** will be used (**\$fp** is not needed)
- ❖ Must save register **\$ra** before making a function call
  - ✧ Must save **\$s0-\$s7** if their values are going to be modified
  - ✧ Other registers can also be preserved (if needed)
  - ✧ Additional space for local variables can be allocated (if needed)

# Steps for Function Call and Return

- ❖ To make a function call ...
  - ✧ Make sure that register **\$ra** is saved before making a function call
  - ✧ Pass arguments in registers **\$a0** thru **\$a3**
  - ✧ Pass additional arguments on the stack (if needed)
  - ✧ Use the **JAL** instruction to make a function call (**JAL** modifies **\$ra**)
- ❖ To return from a function ...
  - ✧ Place the function results in **\$v0** and **\$v1** (if any)
  - ✧ Restore all registers that were saved upon function entry
    - Load the register values that were saved on the stack (if any)
  - ✧ Free the stack frame: **\$sp = \$sp + N** (stack frame = **N** bytes)
  - ✧ Jump to the return address: **jr \$ra** (return to caller)

# Preserving Registers

- ❖ The MIPS software specifies which registers must be preserved across a function call, and which ones are not

Must be Preserved	Not preserved
Return address: <b>\$ra</b>	Argument registers: <b>\$a0</b> to <b>\$a3</b>
Stack pointer: <b>\$sp</b>	Value registers: <b>\$v0</b> and <b>\$v1</b>
Saved registers: <b>\$s0</b> to <b>\$s7</b> and <b>\$fp</b>	Temporary registers: <b>\$t0</b> to <b>\$t9</b>
Stack above the stack pointer	Stack below the stack pointer

- ❖ Caller saves register **\$ra** before making a function call
- ❖ A callee function must preserve **\$sp**, **\$s0** to **\$s7**, and **\$fp**.
- ❖ If needed, the caller can save argument registers **\$a0** to **\$a3**. However, the callee function is free to modify them.

# Example on Preserving Register

- ❖ A function **f** calls **g** twice as shown below. We don't know what **g** does, or which registers are used in **g**.
- ❖ We only know that function **g** receives two integer arguments and returns one integer result. Translate **f**:

```
int f(int a, int b) {  
    int d = g(b, g(a, b));  
    return a + d;  
}
```

# Translating Function f

```
int f(int a, int b) {  
    int d = g(b, g(a, b)); return a + d;  
}
```

```
f: addiu    $sp, $sp, -12      # allocate frame = 12 bytes  
    sw      $ra, 0($sp)      # save $ra  
    sw      $a0, 4($sp)      # save a (caller-saved)  
    sw      $a1, 8($sp)      # save b (caller-saved)  
    jal     g                 # call g(a,b)  
    lw      $a0, 8($sp)      # $a0 = b  
    move    $a1, $v0         # $a1 = result of g(a,b)  
    jal     g                 # call g(b, g(a,b))  
    lw      $a0, 4($sp)      # $a0 = a  
    addu    $v0, $a0, $v0    # $v0 = a + d  
    lw      $ra, 0($sp)      # restore $ra  
    addiu   $sp, $sp, 12     # free stack frame  
    jr     $ra               # return to caller
```

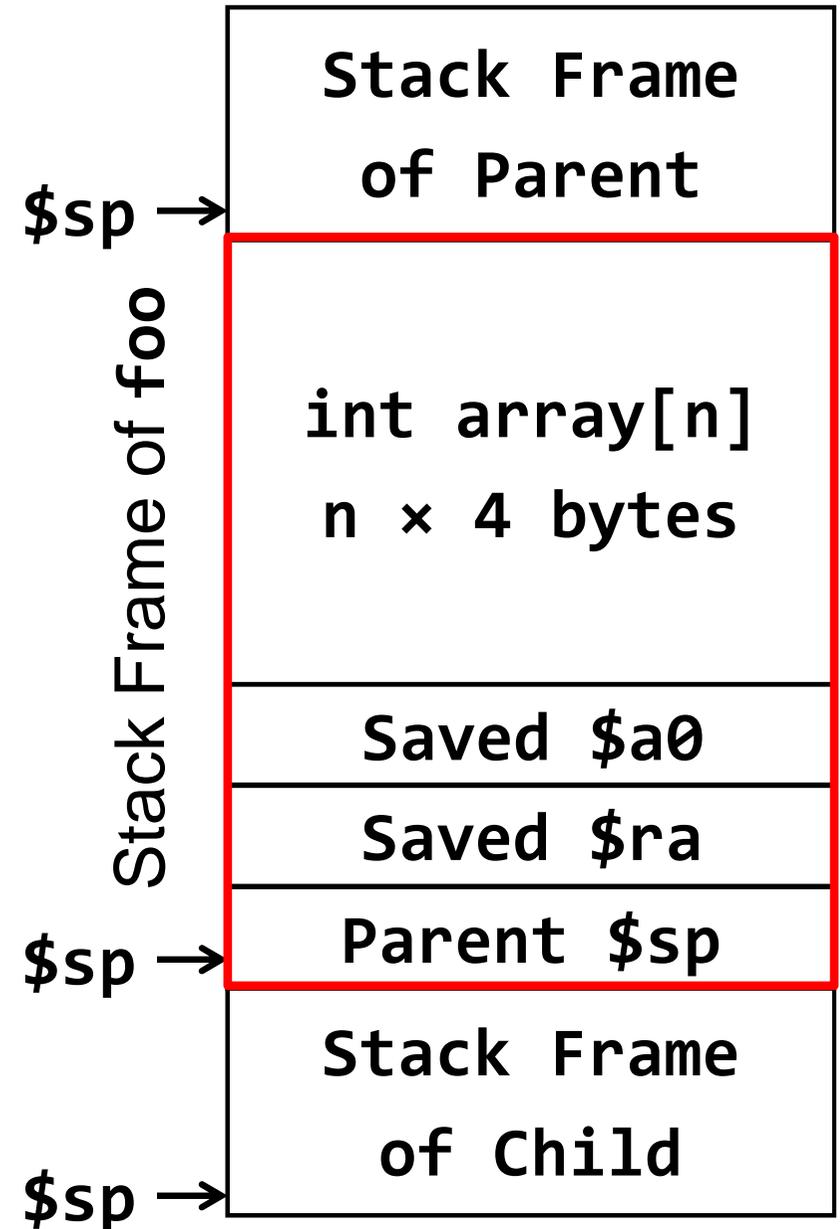
# Next ...

- ❖ Functions
- ❖ Function Call and Return
- ❖ The Stack Segment
- ❖ Preserving Registers
- ❖ **Allocating a Local Array on the Stack**
- ❖ **Examples: Bubble Sort and Recursion**

# Allocating a Local Array on the Stack

- ❖ In some languages, an array can be allocated on the stack
- ❖ The programmer (or compiler) must allocate a stack frame with sufficient space for the local array

```
void foo (int n) {  
    // allocate on the stack  
    int array[n];  
    // generate random array  
    random (array, n);  
    // print array  
    print (array, n);  
}
```



# Translating Function foo

```
foo:                                     # $a0 = n
    sll    $t0, $a0, 2                   # $t0 = n*4 bytes
    addiu  $t0, $t0, 12                  # $t0 = n*4 + 12 bytes
    move   $t1, $sp                      # $t1 = parent $sp
    subu   $sp, $sp, $t0                 # allocate stack frame
    sw     $t1, 0($sp)                   # save parent $sp
    sw     $ra, 4($sp)                   # save $ra
    sw     $a0, 8($sp)                   # save n
    move   $a1, $a0                       # $a1 = n
    addiu  $a0, $sp, 12                   # $a0 = $sp + 12 = &array
    jal    random                         # call function random
    addiu  $a0, $sp, 12                   # $a0 = $sp + 12 = &array
    lw     $a1, 8($sp)                   # $a1 = n
    jal    print                          # call function print
    lw     $ra, 4($sp)                   # restore $ra
    lw     $sp, 0($sp)                   # restore parent $sp
    jr     $ra                            # return to caller
```

# Remarks on Function foo

- ❖ Function starts by computing its frame size:  $\$t0 = n \times 4 + 12$  bytes
  - ✧ Local array is  $n \times 4$  bytes and the saved registers are 12 bytes
- ❖ Allocates its own stack frame:  $\$sp = \$sp - \$t0$ 
  - ✧ Address of local stack array becomes:  $\$sp + 12$
- ❖ Saves parent  $\$sp$  and registers  $\$ra$  and  $\$a0$  on the stack
- ❖ Function **foo** makes two calls to functions **random** and **print**
  - ✧ Address of the stack array is passed in  $\$a0$  and  $n$  is passed in  $\$a1$
- ❖ Just before returning:
  - ✧ Function **foo** restores the saved registers: parent  $\$sp$  and  $\$ra$
  - ✧ Stack frame is freed by restoring  $\$sp$ :  $lw \$sp, 0(\$sp)$

# Bubble Sort (Leaf Function)

```
void bubbleSort (int A[], int n) {
    int swapped, i, temp;
    do {
        n = n-1;
        swapped = 0;           // false
        for (i=0; i<n; i++) {
            if (A[i] > A[i+1]) {
                temp = A[i];   // swap A[i]
                A[i] = A[i+1]; // with A[i+1]
                A[i+1] = temp;
                swapped = 1;   // true
            }
        }
    } while (swapped);
}
```

Worst case Performance  $O(n^2)$

Best case Performance  $O(n)$

# Translating Function Bubble Sort

```
bubbleSort:                # $a0 = &A, $a1 = n
do:  addiu   $a1, $a1, -1   # n = n-1
     blez    $a1, L2       # branch if (n <= 0)
     move    $t0, $a0      # $t0 = &A
     li     $t1, 0         # $t1 = swapped = 0
     li     $t2, 0         # $t2 = i = 0
for:  lw     $t3, 0($t0)    # $t3 = A[i]
     lw     $t4, 4($t0)    # $t4 = A[i+1]
     ble    $t3, $t4, L1   # branch if (A[i] <= A[i+1])
     sw     $t4, 0($t0)    # A[i] = $t4
     sw     $t3, 4($t0)    # A[i+1] = $t3
     li     $t1, 1         # swapped = 1
L1:  addiu   $t2, $t2, 1    # i++
     addiu   $t0, $t0, 4    # $t0 = &A[i]
     bne    $t2, $a1, for   # branch if (i != n)
     bnez   $t1, do        # branch if (swapped)
L2:  jr     $ra            # return to caller
```

# Example of a Recursive Function

```
int recursive_sum (int A[], int n) {  
    if (n == 0) return 0;  
    if (n == 1) return A[0];  
    int sum1 = recursive_sum (&A[0], n/2);  
    int sum2 = recursive_sum (&A[n/2], n - n/2);  
    return sum1 + sum2;  
}
```

## ❖ Two recursive calls

- ✧ First call computes the sum of the first half of the array elements
- ✧ Second call computes the sum of the 2<sup>nd</sup> half of the array elements

## ❖ How to translate a recursive function into assembly?

# Translating a Recursive Function

```
recursive_sum:           # $a0 = &A, $a1 = n
    bnez    $a1, L1      # branch if (n != 0)
    li     $v0, 0
    jr     $ra           # return 0
L1:  bne    $a1, 1, L2   # branch if (n != 1)
    lw     $v0, 0($a0)   # $v0 = A[0]
    jr     $ra           # return A[0]
L2:  addiu  $sp, $sp, -12 # allocate frame = 12 bytes
    sw     $ra, 0($sp)   # save $ra
    sw     $s0, 4($sp)   # save $s0
    sw     $s1, 8($sp)   # save $s1
    move   $s0, $a0      # $s0 = &A (preserved)
    move   $s1, $a1      # $s1 = n (preserved)
    srl   $a1, $a1, 1    # $a1 = n/2
    jal   recursive_sum  # first recursive call
```

# Translating a Recursive Function (cont'd)

```
srl    $t0, $s1, 1           # $t0 = n/2
sll    $t1, $t0, 2          # $t1 = (n/2) * 4
addu   $a0, $s0, $t1        # $a0 = &A[n/2]
subu   $a1, $s1, $t0        # $a1 = n - n/2
move   $s0, $v0             # $s0 = sum1 (preserved)
jal    recursive_sum        # second recursive call
addu   $v0, $s0, $v0        # $v0 = sum1 + sum2
lw     $ra, 0($sp)          # restore $ra
lw     $s0, 4($sp)          # restore $s0
lw     $s1, 8($sp)          # restore $s1
addiu  $sp, $sp, 12         # free stack frame
jr     $ra                  # return to caller
```

❖ **\$ra**, **\$s0**, and **\$s1** are preserved across recursive calls

# Illustrating Recursive Calls

