# MIPS Functions and the Runtime Stack

Computer Architecture

Riad Bourbia

Computer Sciences department

Guelma University

[Adapted from slides of Dr. Mudawar, & El-maleh]

# 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 | | | | | |
|---|---|---|---|---|---|---|---|
| `jal label` | `$31 = PC+4, j Label` | Op=3 | 26-bit address | | | | |
| `jr Rs` | `PC = Rs` | Op=0 | Rs | 0 | 0 | 0 | 8 |
| `jalr Rd, Rs` | `Rd = PC+4, PC = Rs` | Op=0 | Rs | 0 | Rd | 0 | 9 |

# Example

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

```c
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)

**Registers**

| | |
|---|---|
| | . . . |
| $a0=$4 | addr a |
| $a1=$5 | 10 |
| | . . . |
| $ra=$31 | ret addr |

**Caller**

```
  la    $a0, a
  li    $a1, 10
  jal   swap
# return here
  . . .
```

```
swap:
 sll $t0,$a1,2
 add $t0,$t0,$a0
 lw  $t1,0($t0)
 lw  $t2,4($t0)
 sw  $t2,0($t0)
 sw  $t1,4($t0)
 jr  $ra
```

# 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 |
| | | swap: |
| 0040003C | sll $8, $5, 2 | 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<<2

0x10000f << 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;
}
```

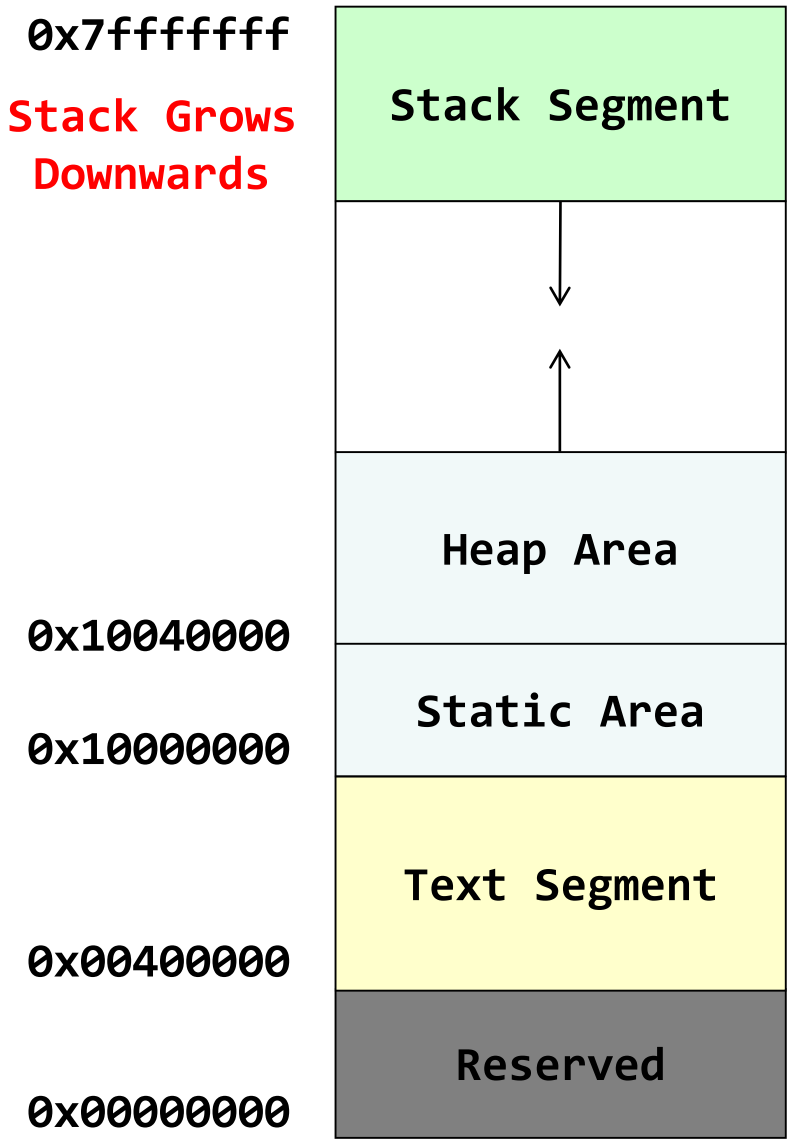
```
tolower:                          # $a0 = parameter ch
  blt   $a0, 'A', else            # branch if $a0 < 'A'
  bgt   $a0, 'Z', else            # branch if $a0 > 'Z'
  addi  $v0, $a0, 32              # 'a' – 'A' == 32
  jr    $ra                       # return to caller
else:
  move  $v0, $a0                  # $v0 = ch
  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

# 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

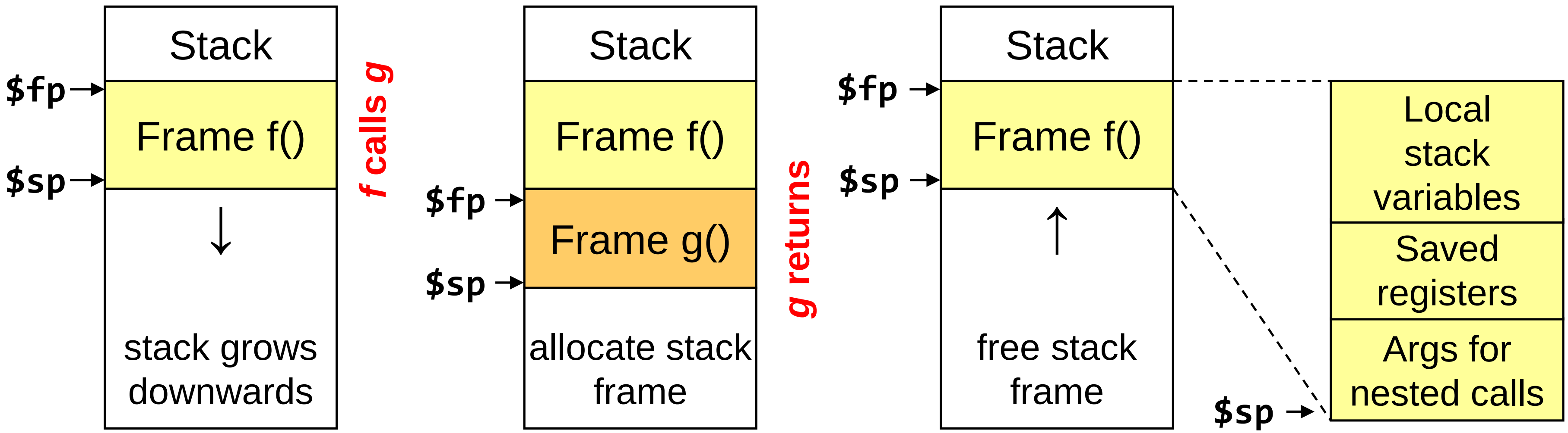


# The Stack Segment (cont'd)

- 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 **`0x7fffeffc`**

# 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



# 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: **$ra** | Argument registers: **$a0** to **$a3** |
| Stack pointer: **$sp** | Value registers: **$v0** and **$v1** |
| Saved registers: **$s0** to **$s7** and **$fp** | Temporary registers: **$t0** to **$t9** |
| 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
```

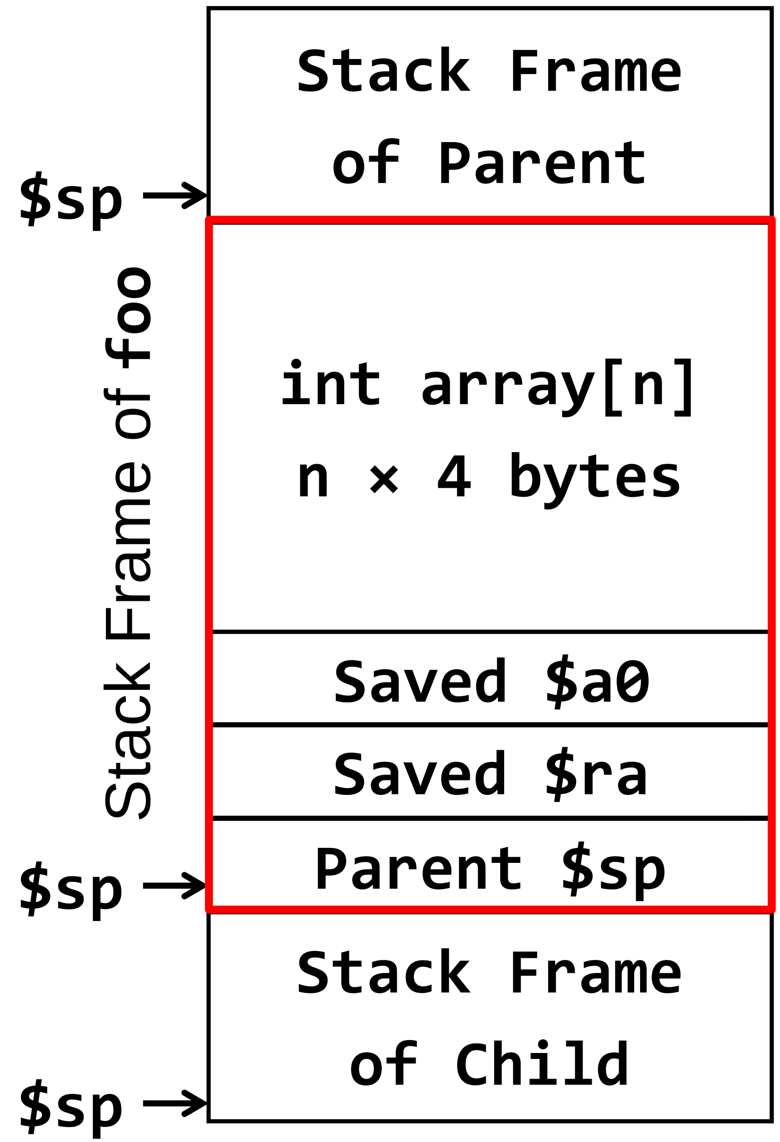
# Next . . .

- Functions
- Function Call and Return
- The Stack Segment
- Preserving Registers
- **Allocating a Local Array on the Stack**

# 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: **`$t0 = n×4 + 12`** bytes
  - Local array is **`n×4`** bytes and the saved registers are **`12`** bytes
- Allocates its own stack frame: **`$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)`**