7 MIPS Functions and the Stack Segment

7.1 Objectives

After completing this lab, you will:

- Write MIPS functions, pass parameters, and return results
- Understand the stack segment, allocate, and free stack frames
- Understand the MIPS register usage convention
- Write recursive functions in MIPS

7.2 MIPS Functions

A function (or a procedure) is a tool that programmers use to structure programs, to make them easier to understand, and to allow the function's code to be reused. A function is a block of instructions that can be called and used when required at several different points in the program.

The function that initiates the call to another function is known as the **caller**. The function that receives and executes the call is known as the **callee**. When the **callee** function finishes execution, control is transferred back to the **caller** function.

A function can receive parameters and return results. The parameters and results act as an interface between a function and the rest of the program.

To execute a function, the program must follow these steps:

1. The **caller** must put the parameters in a place where the **callee** function can access them
2. Transfer control to the **callee** function
3. Execute the **callee** function
4. The **callee** function must put the results in a place where the **caller** can access them
5. Return control to the **caller** (point of origin) next to where the call was made

Registers are the fastest place to pass parameters and return results. The MIPS architecture follows the following software conventions for passing parameters and returning results:

- **`$a0-$a3`**: four argument registers in which to pass parameters
- **`$v0-$v1`**: two value registers in which to return function results
- **`$ra`**: one return address register to return back to the caller

The **jal** (jump-and-link) instruction initiates the call to a function and the **jr** (jump register) instruction returns control back to the caller.

To call a function, use the **jal** instruction as follows:

```
jal label
```

The **jal** instruction saves the return address in register **$ra** and jumps to the first instruction in the function after **label**. The return address is the address of the next instruction that appears after the **jal** instruction in the caller function.

To return from a function, use the **jr** instruction as follows:

```
jr $ra
```

The **jr** instruction jumps to the address stored in **$ra**. It modifies the program counter **PC** register according to the value stored in register **$ra**.

An example of a C function that checks whether a character **ch** is a lowercase letter or not is shown in Figure 7.1. The function is translated into MIPS assembly language as shown to the right. The function **islower** assumes that the parameter **ch** is passed in register **$a0**. The function result is passed in register **$v0**.

```
int islower(char ch) {
  if (ch>='a' && ch<='z')
    return 1;
  else
    return 0;
}
```

```
islower:
  blt $a0, 'a', else  # branch if $a0 < 'a'
  bgt $a0, 'z', else  # branch if $a0 > 'z'
  li  $v0, 1          # $v0 = 1
  jr  $ra             # return to caller
else:
  li  $v0, 0          # $v0 = 0
  jr  $ra             # return to caller
```

Figure 7.1: Example of a C function and its translation into MIPS assembly code

To call the function **islower**, the caller must first copy the character **ch** into register **$a0** and then make the function call. This is shown in Figure 7.2:

```
move $a0, ...         # move into register $a0 the character ch
jal  islower          # call function islower
. . .                 # return here after executing function islower
```

Figure 7.2: Using the **jal** instruction in MIPS to initiate a function call

Remember that the **jal** instruction saves the return address in register **$ra** and that **jr** jumps into the return address in register **$ra** to achieve a function return.

The MIPS architecture provides three instructions to support functions and methods in high-level programming languages. The **jal** (jump-and-link) instruction is used to call functions whose addresses are constants known at compile time, while the **jalr** (jump-and-link register) instruction

is used to call methods whose addresses are variables known at runtime. The **`jr`** (jump register) instruction can be used to return from function calls and methods. These instructions, their meaning, and format are summarized in Figure 7.3.

Instruction	Meaning	Format					
jal label	$ra=PC+4, jump	$op^6 = 3$	imm^{26}				
jr Rs	PC = Rs	$op^6 = 0$	rs^5	0	0	0	8
jalr Rd, Rs	Rd=PC+4, PC=Rs	$op^6 = 0$	rs^5	0	rd^5	0	9

Figure 7.3: The **`jal`**, **`jr`**, and **`jalr`** instructions in MIPS

7.3 The Stack Segment and the Stack Pointer Register

Every program has three segments when it is loaded into memory by the operating system. There is the **text segment** where the machine language code is stored, the **data segment** where space is allocated for constants and variables, and the **stack segment** that 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.

When a program is loaded into memory, the operating system initializes the stack pointer **`$sp`** (register **`$29`**) to the base address of the stack segment. The stack segment grows downwards towards lower memory addresses as shown in Figure 7.4.

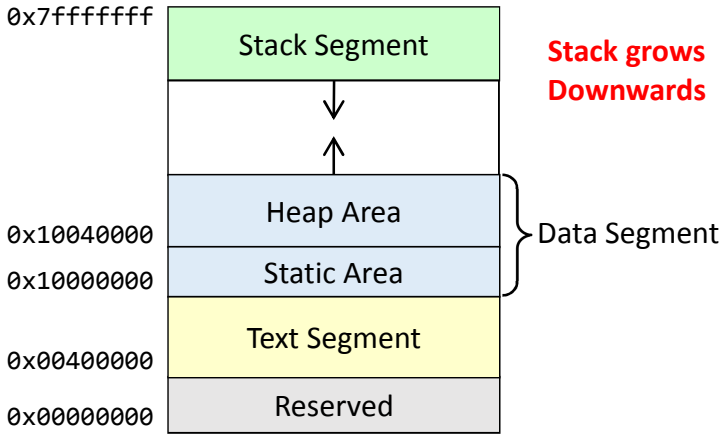

Figure 7.4: The text, data, and stack segments of a program

When a program starts execution, the operating system initializes the stack pointer **`$sp`** register with a valid address to point to the **top of the stack**. For example, when executing a MIPS program under the MARS tool, the initial value of register **`$sp`** is **`0x7fffeffc`**.

A function can allocate space on the stack for saving registers and for its local variables. The space that a function allocates on the stack is called a **stack frame** (called also an **activation record**).

To allocate a stack frame of **n** bytes, decrement the stack pointer by **n** at the start of a function:

```
addiu $sp, $sp, -n    # n must be a constant number of bytes
```

To free a stack frame of **n** bytes, increment the stack pointer by **n** just before a function return:

```
addiu $sp, $sp, n     # n must be a constant number of bytes
```

Figure 7.5 illustrates the stack allocation before calling a function, during the execution of function, and after returning from a function call. The stack pointer register **$sp** points to the top of the caller's stack frame before making a call to function **f**. The **$sp** register points to the stack frame of function **f** during its execution. The **$sp** register points back to the top of the caller's stack frame after returning from function **f**. The stack frame can be used to pass arguments to a function, to save registers across function calls, and to allocate space for local variables declared inside the function. In particular, register **$ra** should be saved before a function can call another function, because the **jal** instruction modifies the **$ra** register. Arguments are typically passed in registers **$a0** thru **$a3**. However, if a function has more than four arguments then the additional arguments should be passed on the stack.

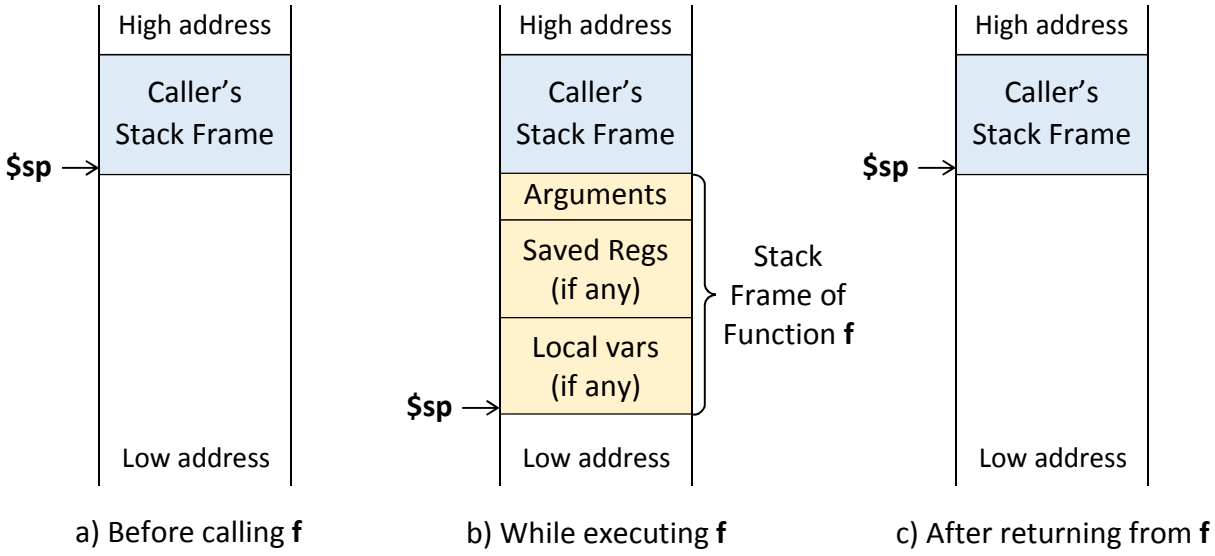

Figure 7.5: Stack allocation (a) before (b) while executing, and (c) after returning from function **f**

An example of a function **f** that allocates a stack frame is shown in Figure 7.6. The function **f** is non-leaf, because it calls functions **read**, **reverse**, and **print**. Therefore, the return address of function **f** (register **$ra**) must be saved on the stack. In addition, the stack frame of function **f** must allocate space for the local array (10 integer elements = 40 bytes), as shown in Figure 7.6.

Example function	Stack Frame

`void f() {`
`  int array[10];`
`  read(array, 10);`
`  reverse(array, 10);`
`  print(array, 10);`
`}`
	`saved $ra = 4 bytes`
	
	`int array[10]`
	`(40 bytes)`

Figure 7.6: Example of a function **f** and its corresponding stack frame

The translation of function **f** into MIPS assembly code is shown in Figure 7.7. Function **f** allocates a stack frame of 44 bytes. The stack is accessed using the same load and store instructions used to access the data segment. The base address register is **$sp**. A displacement is used to access different elements on the stack.

```
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
```

Figure 7.7: Translation of function **f** into MIPS assembly code

Some MIPS software uses the frame pointer register **$fp** (register **$30**) to point to the base address of a stack frame. This might be needed if the stack pointer **$sp** changes during the execution of a function, or arrays and objects are allocated dynamically on the stack. The frame pointer **$fp** register provides a stable address for a stack frame within a function.

7.4 MIPS Register Usage

A convention regarding the usage of registers is necessary because software is written by many programmers. In this case, each programmer must know how registers are supposed to be used, such that his piece of the software does not conflict with pieces written by other programmers.

Since programming is done today using high-level programming languages, you may ask why such a register convention is still needed. Well, it is the compiler who needs to know about it. This is because a program can be created from different pieces that are compiled separately. To compile a function, the compiler must know which registers are used to pass parameters, which registers are used to return results, and which registers must be preserved across function calls. These rules for **register usage** are also known as **function call conventions**.

The MIPS hardware does not prevent you from ignoring these rules, from not preserving registers, from using any register in passing parameters and returning results. However, if you ignore these rules, you will easily run into trouble and have software bugs that are difficult to eliminate.

The following table presents the MIPS register usage convention:

Register Name	Register Number	Register Usage
$zero	**$0**	Always zero. Cannot be modified
$at	**$1**	Reserved for assembler use
$v0 - $v1	**$2 - $3**	Function results are returned in **$v0** and **$v1**
$a0 - $a3	**$4 - $7**	Function arguments are passed in **$a0** thru **$a3**
$t0 - $t7	**$8 - $15**	Temporary registers. Not preserved across function calls
$s0 - $s7	**$16 - $23**	Saved registers. Must be preserved across function calls
$t8 - $t9	**$24 - $25**	Additional temporary registers. Not preserved
$k0 - $k1	**$26 - $27**	Reserved for OS kernel usage
$gp	**$28**	Global pointer to global data. Must be preserved
$sp	**$29**	Stack pointer. Must be preserved
$fp	**$30**	Frame pointer. Must be preserved
$ra	**$31**	Return address register. Must be preserved

Figure 7.8: MIPS register usage convention

A function is free to modify the value registers **$v0-$v1**, the argument registers **$a0-$a3**, and the temporary registers **$t0-$t7** and **$t8-$t9** without saving their old values. However, it should not modify registers **$s0-$s7**, **$gp**, **$sp**, **$fp**, and **$ra** except after saving their old values in memory on the stack. A function must restore the value of registers **$s0-$s7**, **$gp**, **$sp**, **$fp**, and **$ra** by loading their old values from the stack, just before returning back to the caller. Registers **$sp** and **$fp** must be preserved if a new stack frame is allocated by a function. Register **$ra** must be preserved if a function makes a call to another function, because the **jal** instruction modifies the return address register **$ra**.

7.5 Recursive Functions

A recursive function is a function that calls itself. For example, the recursive function **fact** (factorial) and its translation into MIPS assembly code are shown in Figure 7.9. If **(n<2)** then there is no need to allocate a stack frame. However, if **(n>=2)** then the factorial function allocates a stack frame of 8 bytes to save registers **$a0** and **$ra**.

Register **$a0** (argument **n**) is saved on the stack because its value is changed in the recursive call, and because it is needed after returning from the recursive call. Register **$ra** is saved on the stack because its value is changed by the recursive call (**jal fact**).

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

```
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 a stack frame of 8 bytes
  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
```

Figure 7.9: A recursive function and its translation into MIPS assembly language code

7.6 In-Lab Tasks

1. The function **islower**, shown in Figure 7.1, tests whether a character **ch** is lowercase or not. Write the **main** function of a program that reads a character **ch**, calls the function **islower**, and then prints a message to indicate whether **ch** is a lowercase character or not.
2. Write a function that reads an array of **n** integers. The function **read** must receive two arguments: **$a0** = address of the array, and **$a1** = number **n** of elements to read.
3. Write a function that prints an array of **n** integers. The function **print** must receive two arguments: **$a0** = address of the array, and **$a1** = number **n** of elements to print.
4. Write a function that reverses the elements of an array of **n** integers. The function **reverse** must receive two arguments: **$a0** = address of the array, and **$a1** = number **n** of elements.
5. Suppose we rewrite function **f** (Figures 7.6) to have an integer parameter **n**. The local **array** is now declared to have **n** integers (rather than **10**). This means that the size of the stack frame size

of function **f** will depend on **n**. Rewrite the function **f** in MIPS assembly language. Hint: you may use the **$fp** register (in addition to **$sp**) to implement the function **f**.

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

6. The function **f(n)** implemented in problem 5 calls the functions **read**, **reverse**, and **print** implemented in problems 2 to 4. Write a complete program that includes the **main** function as well as functions **f**, **read**, **reverse**, and **print**. The **main** function should call function **f** twice as: **f(5)** and **f(8)**.

7. The recursive function **fib(n)** computes the n^{th} element in the Fibonacci sequence. Implement this function in MIPS. Write a **main** function to call **fib**.

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

7.7 Bonus Question

8. The function **quick_sort** sorts an **array** recursively. Translate this function into MIPS code. Write a main function to call and test this function.

```
void quick_sort(int array[], int low, int high) {
  int i = low, j = high;              // low and high index
  int pivot = array[(low+high)/2];    // pivot = middle value
  while (i <= j) {
    while (array[i] < pivot) i++;
    while (array[j] > pivot) j--;
    if (i <= j) {
      int temp=array[i];
      array[i]=array[j];              // swap array[i]
      array[j]=temp;                  // with array[j]
      i++;
      j--;
    }
  }
  if (low < j)  quick_sort(array, low, j);    // Recursive call 1
  if (i < high) quick_sort(array, i, high);   // Recursive call 2
}
```