Lab 07 report: MIPS Functions and the Stack Segment

Objectives

- To learn how write MIPS functions, pass parameters, and return results.
- To understand the stack segment, allocate, and free stack frames.
- To understand the MIPS register usage convention.
- To learn how to implement recursive functions in MIPS.

Introduction

A function is a tool that developers 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** function such as the main function. The function that receives and executes the call is known as the **callee** function. When the callee function finishes execution, control is transferred back to the caller function. A function can receive parameters using `($a0 →$a3)` registers and return results through `($v0 and $v1)` registers. The parameters and results act as an interface between a function and the rest of the program.

Tasks

Task1:

1. **Requirement:** it is required to implement a MIPS program that implements the **`read`**, **`reverse`** and **`print`** functions as shown in **figure.1** used by f function as shown in **figure.2** considering that these functions should be compatible with any size of integer n including the main function.

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.1: Example of a function `f` and its corresponding stack frame

```
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.2: Translation of function `f` into MIPS assembly code

2. **Approach:** to implement this program it is required to write a `main` function that calls the `f` function using `jal f` instruction which will execute `f` function.

To implement the read function, we need to specify where the parameters are stored. We can see from `f` function that the array address is stored in `$sp` register which is stored in `$a0` and n which is stored in `$a1` register. Then we should preserve `$sp` address by moving it into a temporary register such as `$t2` because we will prompt the user to enter the integer numbers using `$a0`. We also stored n into another register `$t0` and initialized `$t1` to 1 to implement the loop which is used to prompt the user to enter the integer numbers using `(syscall 5)` and store them in the array. The instruction that will determine the loop iteration number is (`bgt $t1, $t0, exitPromptLoop`) which will start from 1 to n. when `$t1` exceeds n it will exit the loop. After we have read the integers from the user, we will store it using `(sw $v0, 0($t2))` in the first address in the array. Then we will increment the address by 4 to move to the next word in the array. Then we will increment `$t1` by 1 followed by `(j promptLoop)` to proceed to the next iteration. The loop will read the integers from the user until it exceeds n. After it exceeds n it will return to `f` function using `(jr $ra)` where `$ra` register contains the next instruction address in the text segment.

To implement reverse function, we need to store n and the address of the array as done in the read function. Then we need to divide n by 2 to determine the swapping execution number. we need also to initialize a register to start a loop to control the swap execution number. then we will multiply n by 4 (where 4 indicates word data type) and store it into another register to add it with the first index to the array which will be as same as (`array[n]` in Java); however, it. also should be stored into another register because we need `arr[i]` to be swapped with `array[(n - 1) - i].` Then we start the loop which will do the main job which its execution condition in Java is `for(i = 1;i < n/2; i++)` which is equivalent to (`li $t1, 1, bgt $t1, $t3, exitReverseLoop, addiu $t1, $t1, 1`). The first thing to consider is adding 4n to the first address is equivalent to writing (array[n] in Java) so we need decrement the address by 4 using (`addiu $t5, $t5, -4`) to make it equivalent to (`array[n – 1]` in Java). We have included it in the loop to move to the previous element in the array until it rechs the middle of the array. Then we will store the first element in the array into a temporary register such as `$t6` using `(lw $t6, 0($t2))` instruction. We need to store the first cell into a temporary register because we will using it later by storing it in the last element in the array. Then we will store the last element of the array into another temporary register such as `$t7` using `(lw $t7, 0($t5))` instruction to store it later in the first element in the array. Then we store the loaded element from the last element into the first element in the array and vice versa. Then we need to increment the address of the first element by 4 to move to the next element. We increment the address by 4 because we are dealing with word data type. Then we increment `$t1` by 1 followed by (`j reverseLoop`) to proceed to the next iteration. The loop will reverse the swap the array until it exceeds n /2. After it exceeds n /2 it will return to `f` function using `(jr $ra)` where `$ra` register contains the next instruction address in the text segment.

To implement the print function we need to store `$a0` value into a temporary register such as `$t2` because we will use $a0 to print the values and spaces. We print the result message using `(syscall 4)` instruction. Then we will construct a loop that will start from 1 to n to print each array element. The condition of the loop is `(bgt $t1, $t0, exitPrintLoop)` where `$t1` is initialized to 1 and `$t0` is initialized to n. then we will store the array element `int $a0` using `( lw $a0, 0($t2))` instruction to print it using (`syscall 1).` Then we will use `(li $a0, ' ')` instruction to print spaces between the integers using `(syscall 11).` Then we will increment the array address by 4 to move to the next element of the array. Then we increment `$t1` by 1 followed by `(j reverseLoop)` to proceed to the next iteration. The loop will reverse the swap the array until it exceeds n. After it exceeds n it will return to `f` function using `(jr $ra)` where `$ra` register contains the next instruction address in the text segment.

Finally, the program will return from `f` function to the `main` function to be terminated using `(syscall 11).`

Tasks

Task2

1. **Requirement:** it is required to implement a MIPS program that compute the Fibonacci number at the index n that is prompted to the user.
2. **Approach:** to implement this program it is required to write a `main` function that prompt the user to enter any non-negative integer n then using `(syscall 5).` After receiving the number, we need to store the value of `$v0` into a temporary register then check if the number is a non-negative using `(bgez $v0, excuteFib)` instruction. If `$v0` is negative it will restart the program, else it will execute `fib` function using `(jal fib)` instruction. But before `jal` instruction we need to move the value of `$v0` into `$a0` to make n as a parameter for `fib` function. When we call `fib` function we need to check if `$a0` is less than 3. If `$a0` is less than 3, `fib` function will return 1 stored in `$v0` using `(jr $ra)` instruction. On the other hand, if `$a0` is grater than or equal to 3 it will jump to `else` label. In `else` block the first thing to do is to allocate 3 words in the stack segment to store `$ra` and (previous n) that is in `$s0` and `$v0` (result of `fib(n)`) that is in `$s1` using (`addi $sp, $sp, -12`) instruction. We take the value of `$a0` which it is n and store it in `$s0` to make `$s0` equal to n then subtract it by 1 and store it in `$a0` to make `$a0` equal to n -1 to call `fib(n -1)` using `(jal fib)` instruction and store the result into `$s1`. Then it will calculate `fib ((n – 1) – 1)` and `fib ((n -1) -2)` and store it in the stack segment until it reaches the base case. It also works `with fib(n -2)` part. After it reaches the base case it will add the final result into `$v0` by adding `$v0` (`fib(n -2)` call) and `$s1` (`fib(n-1)` call) using (`add $v0, $s1, $v0`) instruction. Then we retrieve the previous values of `$ra`, `$s0` and `$s1` using `(lw $ra, 0($sp)), (lw $s0, 4($sp) )` and `(lw $s1, 8($sp))` instructions. Then we empty the stack segment from the retrieved values using `(addi $sp, $sp, 12)` instruction. Finally, we return to the previous caller using `(jr $ra)` instruction until we reach the main function to print the result using `(syscall 1)` from `$v0` by storing `$v0` value into `$a0` register.

Conclusion

In conclusion, MIPS offers the possibility to implement functions using jump and link and jump and return which allow the code to be reused in multiple times without the need to repeat it. MIPS also offers the possibility to implement recursive functions with the help of the stack segment and pointer register to store the values of the previous call and its address.