LAB #06 Procedure Call

MIPS Procedure

A procedure (or function) 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

`jal` (jump-and-link) instruction initiates the call to a function.
`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 Fig 1. The function `islower` assumes that the parameter `ch` is passed in register `$a0`. The function result is passed in register `$v0`.

C function	Assembly
	`islower:`
`int islower(char ch) {`	`    blt $a0, 'a', else # branch if $a0 < 'a'`
`  if (ch>='a' && ch<='z')`	`    bgt $a0, 'z', else # branch if $a0 > 'z'`
`    return 1;`	`    li  $v0, 1         # $v0 = 1`
`  else`	`    jr  $ra            # return to caller`
`    return 0;`	`else:`
`}`	`    li  $v0, 0         # $v0 = 0`
	`    jr  $ra            # return to caller`

Fig. 1 Example of a C function and corresponding 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 as follows:

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

Note that the **`jal`** instruction saves the return address in register **`$ra`** and **`jr`** jumps into the return address in register **`$ra`** to achieve a function return.
The MIPS architecture provides three instructions to support functions and methods in high-level programming languages.

`jal` (jump-and-link): used to call functions whose addresses are constants known at compile time.
`jalr` (jump-and-link register): used to call methods whose addresses are variables known at runtime.
`jr` (jump register): can be used to return from function calls and methods.

Instruction	Meaning
`jal  label`	`$ra = PC + 4, jump`
`jr   Rs`	`PC = Rs`
`jalr Rd, Rs`	`Rd = PC + 4, PC = Rs`

Fig. 2 `jal`, `jr`, and `jalr` instructions in MIPS

Example of procedure call

In class, we discussed frame pointer($fp), stack pointer($sp), and how they point to make a space for a new procedure, etc., so we will not discuss them again here. (Check the content around figure 2.12 in the textbook, or chapter 2 lecture slides p.48).

An example of a function **f** that allocates an activation record (or procedure frame) is as follows:

```
void f() {
    int array[10];
    read(array, 10);
    reverse(array, 10);
    print(array, 10);
}
```

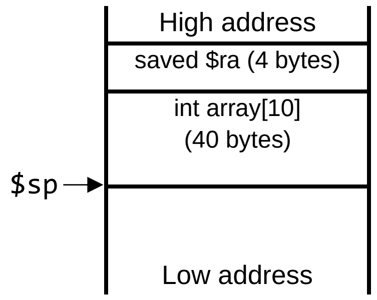

Fig. 3 Example of a non-leaf procedure `f()`

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 activation record of function **f** must allocate space for the local array (10 integer elements = 40 bytes).

The translation of function **f** into MIPS assembly code is shown below.
Function **f** allocates an activation record 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 activation record 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 activation record of 44 bytes
      jr    $ra            # return to caller
```

Fig. 4 MIPS assembly example of a non-leaf procedure `f()`

Task to do

1. The function **islower** (shown in Fig. 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 **fact(n)** which calculates factorial of n (i.e., n!) according to the following C code. Also, write **main** to call **fact**.

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

NOTE:

1. Write a report according to the LAB questions and submit a word (or pdf) file.
2. For each question, **write** assembly code (**not** screen capture… hard to read).
3. For each problem save an asm file and **zip** them with your report.
   (file name of your asm file should be lab.6.1.asm, lab.6.2.asm, etc.)