# Subroutines/Functions in MIPS

## Introduction

Functions are perhaps the most fundamental unit of programming used in all of programming languages. It gives us the simplest form of program abstraction. It provides an interface (i.e., the prototype) and allows us to use the function without knowing how it is implemented.

Thus, it makes sense that assembly languages must provide the mechanism to implement functions. To make a distinction between functions used in programming languages and those used in assembly languages, I will refer to function support as *subroutines*. In any case, there's a difference between functions in, say, C, and subroutines in an assembly language. If I occasionally call them functions, I really mean MIPS subroutines.

There are two ideas behind a subroutine.

- You should be able to call the subroutine from anywhere.
- Once the subroutine is complete, it should return back to the place that called the subroutine.

We'll see how both are done.

## Collaboration

Calling a subroutine involves two participants: the **caller** and the **callee**. The *caller* calls the subroutine. The caller's job is to set up arguments to the subroutine, then jump to the subroutine. At this point, the *callee* take over.

The **callee** is the subroutine that's called by the **caller**. The callee does not have information about what section of code called the callee. It just "knows" it has been called. That's the same as it is in C or C++. When a function is called, it's difficult to tell who called it. In a debugger, you can look at the call stack, but usually, the function itself can't tell.

The **callee** uses the arguments given to it by the **caller** and then performs computations. When it is done, the **callee** places the return value results, and then returns control (i.e., jumps back) to the **caller**.

As the **callee** subroutine code is being executed, the **callee** may call a helping subroutine. So, the callee may become a caller. Thus, the notions of caller and callee aren't always clearly divided.

- **caller** makes the subroutine call
- **callee** is the subroutine code itself, and is called by the callee.

This mechanism shows an agreement (also called a *protocol*) between the **caller** who provides the arguments and **callee** who uses the arguments to do computations and return a value, and finally back to the **caller** who uses the return value.

## Limited Resources

When you program in C or C++ or Java, you are used to calling functions with local variables. Each time you call a function, a new set of local variables is created. This is why recursive function calls work. Each recursive call has its own copy of local variables and parameters (unless the parameters are passed by reference). This makes it easier to write functions in procedural languages.

When you program in assembly language, there is only one set of registers used in the program. In effect, these registers act like global variables. It's very easy to make a subroutine call, and think that after the subroutine call is done, the registers have remain unchanged.

However, you would be wrong if you thought thatw. When you make a subroutine call, you have to assume that, unless convention dictates otherwise, the subroutine will clobber all the registers you are using (except the stack pointer). Thus, if you call a subroutine, any values you've stored in a register *could* be overwritten. After all, the subroutine being called needs to use registers too, and there's only one set to work with.

### You're the Callee

MIPS designates 8 registers **$s0-$s7** as *saved* registers. It's up to the callee to save the registers if they're being used. This is *not* handled automatically by the CPU. It's merely a convention that's followed by MIPS programmers.

This seems like a great idea. Let's say you are writing subroutine FOO, and subroutine **FOO** calls subroutine**BAR**. So, you think "I'm going to use registers **$s0-$s7** because they're saved by the callee, so when I call **BAR**, then **BAR** will save it for me". And therefore, you don't think it's necessary to save the saved registers.

Except that's not the correct view. Any subroutine can serve as the **caller** and the **callee**. In fact, your initial view as a subroutine writer is to imagine you're the callee. After all, you're providing a subroutine for *others* to use (which could be you). It's easy to think of yourself as the **caller**, but think first as a **callee**.

Since you now view yourself as a callee, this is what you need to do:

- Step 1: Determine what saved registers you intend to use
- Step 2: Push those saved registers onto the stack
- Step 3: Use the saved registers and run the main part of the subroutine code.
- Step 4: Just before you return from the subroutine call, pop the saved registers
- Step 5: Return back to the subroutine that called you.

Suppose you're in Step 3. You've pushed the saved registers on the stack, and now you're running code. You decide to call a subroutine. Do you need to resave the saved registers? At this point, you can imagine yourself as the caller, and thus, you don't have to save the registers a second time. In fact, you already saved them in Step 2.

The subroutine you're about to call will save any registers it needs on the stack, which may be different from the registers you saved. It doesn't really matter because it's up to that subroutine to figure it out.

So, the right mindset to writing a subroutine is to think "I'm the callee".

## The Stack

If registers are a shared commodity, and you can't expect subroutines to save the registers, then what can you do to have some section of memory for your subroutine which others won't overwrite?

By convention, each subroutine uses a part of the stack, and it's assumed that each subroutine will only use its part of the stack, for the most part.

Let's see what happens in a call.

- Caller pushes arguments onto the stack (caller stack frame)
- Caller pushes a location for the return value on the stack (caller stack frame)
- Callee accesses arguments in caller's stack frame
- Callee pushes space for local variables (callee stack frame)
- Callee may itself call other subroutines
- When the callee computes the return value, it places it in the caller part of the stack. Remember that the caller reserved some section of the stack for the return value.
- Callee restores stack pointer back to the way it found it just as it was being called. Thus, stack pointer now points to the return value.
- Caller gets the return value, and eventually pops that off and the arguments off.

This view is rather old-fashioned, and is not how MIPS handles subroutine calls. However, it's good to see how it used to be done to compare and contrast.

In MIPS,

- Caller places arguments in registers **$a0-$a3**. It can place up to four arguments. If there are more, or if any of the arguments is a structure being passed by value, then it's placed on the stack.
- Caller does not need to push a location for the return value. Registers **$v0** and **$v1** are used for the return value.
- Callee accesses arguments and return value from these registers.
- Callee pushes space if it needs to save registers (just in case it calls subroutines).
- Callee may itself call other subroutines
- When the callee computes the return value, it places it in register **$v0** (and **$v1**, if needed).
- Callee restores stack pointer back to the way it found it just as it was being called. Thus, stack pointer now points to the return value.
- Caller retrieves return value from **$v0**. If it needs to make another subroutine call, it may need to save the return value on the stack (otherwise, it will be overwritten on the next subroutine call).

Each subroutine, as it is running, will have a part of the stack for its own use. This is called the *stack frame*. By convention, the subroutines just use its part of the stack. The exception is when the callee needs to access arguments passed by the caller. The arguments are considered part of the caller's stack (you could view it as shared too).

It may be possible for a subroutine code to have more than one stack frame. For example, recursive functions will have a stack frame for each recursive call that's made. Subroutines do not need to be recursive for there to be two or more stack frames associated with the subroutine.

For example, you might have a function that computes the minimum of an array called **findMin()**. **findMin()** may get called in **foo()** which then calls **bar()** and **bar()** may itself call **findMin()**, thus **findMin()** has two stack frames on the call stack, even though none of the functions are recursive.

## What Needs to Happen in a Subroutine Call

Let's imagine you need to make a call to a subroutine called **QUICKSORT**.

What do you need to do? If this were, say, C, you would have to pass arguments to the subroutine and then make the call.

Let's see how that might work.

```
.... | # Set up arguments
1000 |    j QUICKSORT    # Call QUICKSORT
.... | ....
.... | ....
2000 | QUICKSORT:  # Code for quicksort
```

In the diagram above, I've written memory addresses on the left side. This shows you were the instructions are stored in memory. Let's assume the call to QUICKSORT is made at address 1000. We do this (for now) by calling "j", which is jump.

The code for quicksort begins at address 2000. Thus, jumping to QUICKSORT means changing the address of the program counter to 2000. Recall the program counter is a hidden register that stores the address of the current instruction.

OK, so far, so good. We've managed to jump to the correct subroutine. QUICKSORT does what it needs to do, and then it needs to jump back.

Except where do we jump back to?

Alas, the program counter does not keep a history of its jumps. When we're in QUICKSORT, we have no idea how we got there. The only place that knows this information is the place where the subroutine was called.

Thus, when we're making the jump at address 1000, we know that's where we need to get back to. Thus, we need to store the information about hot to get back at that point.

that's why we use "jal" instead of "j"

## Making a Subroutine Call

The most important instruction for making a subroutine call is **jal** which means "jump-and-link".

**jal** takes a label as its operand. This label is an address in memory for a subroutine. The assembler translates the label to an address.

To jump to that address really means to update the PC (program counter) to the address of the subroutine. The PC is a hidden register that holds the address of the current instruction being run.

The **jal** instruction saves the return address in register **$r31**. This register is also called **$ra** (where "ra" means return address).

So we modify the code we had before to use **jal** instead of **j**.

```
.... | # Set up arguments
1000 |    jal QUICKSORT    # Call QUICKSORT
.... | ....
.... | ....
2000 | QUICKSORT:  # Code for quicksort
```

How do we know what to save into register 31? It saves the (current addr of jal + 4)

We could save address 1000. That way, once the QUICKSORT subroutine is done, it could go back to the instruction that called it, which is stored at address 1000.

However, that's pretty silly. Address 1000 stores the **jal** instruction, and we'd end up going back to the same subroutine call.

We really want to execute the *next* instruction in memory. Since each MIPS instruction uses 4 bytes, that instruction is at 1000 + 4 = 1004.

So, 1004 is saved in register 31.

### Returning Back to Caller using `jr`

We have **jal** to get to the subroutine call, but once we're in the subroutine call, and we're ready to return, we need to jump back.

Where do we jump back to? To the address in register 31. How do we do that?

There must be a special instruction for jumping to the address stored in a register. And so there is. It's called **jr** for "jump register".

At the end of the subroutine code for QUICKSORT, we need to call **jr $ra**.

```
2000 | QUICKSORT:  # Code for quicksort
```

```
....  |  .....
....  |  .....
20ff  |              jr $ra  # Return back
```

**Problem with Subroutines Calling Helper Subroutines**

You may have already noticed a problem. When we called QUICKSORT using **jal**, the arguments were passed in **$a0-$a3** the return address was stored in **$ra**.

What if QUICKSORT calls a helper subroutine, HELPER?

Let's first define a helper subroutine:

- **Helper subroutines** are any **jal** calls within the subroutine

When QUICKSORT calls HELPER, arguments are going to be passed to HELPER using registers **$a0-$a3** and the **jal** will overwrite the return address.

This is the standard problem with assembly language. Registers are shared by all subroutines. When you make a call to a helper subroutine that subroutine uses the same registers as you do.

The solution? Before you call a subroutine, save the return address and any registers you need on the stack.

**Figuring Out What to Save on the Stack**

As you're writing a subroutine, you should determine what to save on the stack.

Here are the guidelines:

- If the subroutine is a leaf procedure, i.e., it doesn't have a **jal** call, then it's easy. You don't need to save any registers on the stack, except for **$s0-$s7**.
- If the subroutine has a **jal** instruction, then make a list of any registers you are using in the subroutine. You will have to, at the very least, save **$ra**, since **jal** will overwrite that.
- Then ask yourself, do you need to use those registers after the subroutine call is made. The answer is usually yes, so they should be pushed on the stack.
- If there's a possibility of least two subroutine calls being made, then you may need to save the return value after each subroutine call is complete.
- Once you're ready to exit the program, you need to restore the return address, and adjust the stack pointer.

Why do you need to save the return value on the stack? Let's look at an example of a subroutine with two calls to helper subroutines, BAR and CAT:

```
FOO:  ....
      ....
      jal BAR
      move $t0, $v0   # Save return value to t0
      ....
      jal CAT
      add $v0, $v0, $t0   # Uh oh! Error! $t0 may have changed!
```

Suppose you read this code by a programmer. In the first comment, **$v0** is "saved" to a temporary register. The programmer did this knowing that the call to **jal CAT** would overwrite **$v0**. However, it's not really saved.

Why not?

Because the CAT subroutine might overwrite **$t0**! Again, you have to assume that any well-behaving subroutine call may overwrite any register except the saved registers and the stack pointer. A badly behaving subroutine may change the saved registers and the stack pointer, but we'll assume that doesn't happen, otherwise, we're going to have a very difficult time programming.

One solution is to used a saved register, **$s0**, but that means we must use another rule. If you use a saved register, you need to save it to the stack before use. Thus, we need to save **$s0** to the stack. Either way, we need to use the stack.

So, the solution is to do something like:

```
FOO:  ....
      ....
      jal BAR
      sw  $v0, 4($sp)      # Save return value to stack
      ....
      jal CAT
      sw  $t0, 4($sp)      # Get old return value from stack
      add $v0, $v0, $t0    # Fixed problem!
```


We saved the return value to **4($sp)**. Presumably, you will have picked a good location to save it on the stack. The choice of **4($sp)** was arbitrary.

**When to Save and Restore the Stack?**

As a matter of habit, programmers often start a subroutine by saving registers to the stack. It's a reasonably good habit, so that you remember to do it. They restore the stack just before calling **jr $ra**.

As you about to return back to the caller, ask yourself what values really need to be restored off the stack. One can argue either way whether argument registers need to be restored. Usually, you can assume that arguments are allowed to be overwritten.

Certainly, **$ra** needs to be adjusted, as does the stack pointer. If you use a frame pointer (see next section), then you have to worry about restoring the frame pointer too.

You can save yourself a few steps by keeping track of what needs to be restored from the stack. However, it generally doesn't hurt anything (except a few cycles) to restore more registers than needed.

**Return Values are Tricky**

Return values of helper subroutine calls are the trickiest to deal with, especially in the case of calling two (or more) helper subroutine call (e.g., BAR and CAT, from earlier on).

You can't save the return values of BAR initially, because you haven't even called BAR. You have to wait until you reach the part of the code where the call to the helper subroutine is made. After that **jal**, you can save the return value on the stack. Then, you can make the call to the second helper subroutine (say, CAT), and once you get back from that, you can retrieve BAR's return value from the stack.

If there isn't a second helper subroutine like CAT, then there's no need to store the return value of the first helper subroutine (i.e., BAR) on the stack.

## Pascal Functions?

Most of our discussion has been with functions/subroutines that behave similar to C functions. C does not support functions nested inside other functions, as Pascal does. Pascal functions can be nested, and the inner functions have access to the outer function's paramaters.

The stack structure is more complicated. We won't how that works. The point is that it's possible to design a more sophisticated stack than the one we've gone over. We've picked C's stack style because it's simple, and it allows us to write subroutines in assembly language, similar to C.

## Summary

Calling a subroutine is a collaboration between a *caller*, who makes the subroutine call, and the *callee*, which is the subroutine itself.

In MIPS,

- the caller passes arguments to the callee by placing the values into the argument registers **$a0-$a3**.
- the caller calls **jal** followed by the label of the subroutine. This saves the return address in **$ra**.
- the return address is PC + 4, where PC is the address of the **jal** instruction
- If the callee uses a frame pointer, then it usually sets it to the the stack pointer. The old frame pointer must be saved on the stack before this happens.
- The callee usually starts by pushing any registers it needs to save on the stack. If the callee calls a helper subroutine, then it must push **$ra** on the stack. It may need to push temporary or saved registers as well.
- Once the subroutine is complete, the return value is place in **$v0-$v1**.
- The callee then calls **jr $ra** to return back to the caller.