Foothill College
Computer Science Department
CS10 Computer Architecture and Organization
Lab # 5
Calling Functions; The Stack; Recursion

1 Introduction

This week we are looking at the mechanism used on the MIPS CPU to call functions, so that programs can be modularized (i.e. as we can in high-level languages like C and Java). To understand how we do this, we also need to understand the stack frame that MIPS uses. Then, to finish up, we will look at recursion in assembly.

2 Function Calls

- It is time to move on to calling functions and returning functions.
- For example, in a high-level function, this is all done by defining a function with some *parameter variables*:

```
public static int myFunction(int x, char y, double z)
{ ... }
```

  and then elsewhere in the program, by calling this function with some *arguments*:

```
int result= myFunction(23, 'g', usersweight);
```

- Behold, there is more going on here than meets the eye. For example, the function will probably want to define its own variables, and these need to be kept *local*, i.e. not visible to the other functions.
- Somehow, a return value has to come out of the function.
- And, we also want to support recursion, because it can be so useful.
- So before we look at the machine instructions to call and return from functions, let's look at the requirements.

2.1 Requirements to Perform Functions

- Here are the things that we need for functions to work:
  1. The function caller needs to be able to divert the program counter (PC) to the first instruction of the function.
  2. At the end of the function, the function needs to be able to return the PC to the instruction *after* the function call instruction. We call this the **return address**, and it needs to be kept somewhere.
  3. The function caller needs to be able to provide a set of arguments, in a specific order, to the function. The function needs to be able to see these as local variables.
  4. The function needs to be able to create its own local variables which are different to those in the functions which called it.

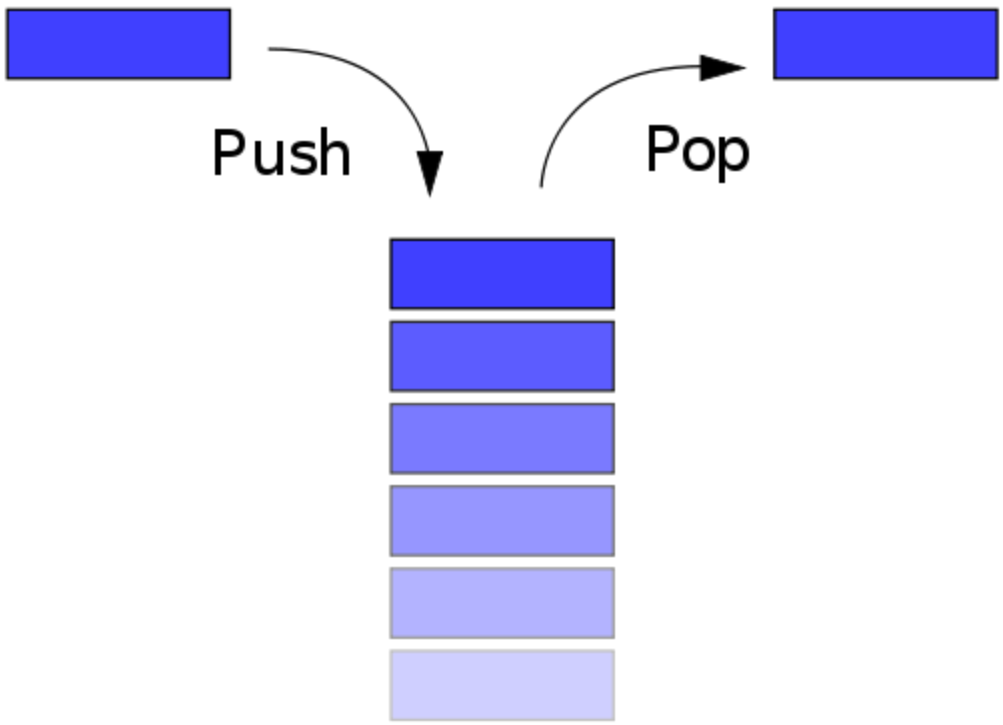
5. Because registers are frequently used, the function caller and the function itself are going to need to agree on how to store away register values, so that each will have access to the CPU's registers without having to worry about losing values.
6. The function needs to be able to return a result to the function caller.
7. We also want recursion: the ability to write functions which call themselves an arbitrary number of times.

2.2 Issues in Performing Functions

- The big problem with implementing function calls on a CPU is that functions can call functions, which can call functions *ad infinitum*.
    - We can not predict the number of function calls.
- Why is this a problem?
    - We need to store arguments to *each* function when we call it, *and*
    - We need to store the return address, so the function knows where to return to, *and*
    - We also need to store registers before a function call, so the function can use them, and so that we can get our own values back when the function returns.
- If there was only 1 function call allowed, we could reserve a fixed amount of global data space for arguments, return address and registers.
- But if functions call other functions, we need a possibly unlimited space for the arguments, return addresses and registers, and we need a way to remember which function saved which registers & return address, and which function received which arguments.

2.3 The Solution: A Stack

- We need some form of dynamically growing data structure, and a mechanism to remember who owns what in this data structure.
- The solution is a **stack**.
- In a stack, items can be pushed on the stack, and items can be popped off the stack.



(from Wikipedia)

- In the strictest sense, items pushed on the stack have to be popped off in reverse order: if we push A, B and C on the stack, then C must be popped off first before we can pop B and get access to it.
- In our implementation of the stack below, we are also able to inspect & modify values already on the stack, i.e. if A, B and C are on the stack, we can look at all three items.
- The stack provides a solution to our problem. Just before each function call, the caller can:
    - save any registers that it wants to keep by pushing them on the stack
    - push the arguments to the function on the stack
    - push the return address on the stack
    - jump to the function's first instruction
- The function now has access to the arguments which are on the stack.
- At the end of the function, the function jumps to the return address which is stored on the stack.
- We are now back in the function caller, which can clean up the stack by popping everything off that it pushed on.
- And if functions call other functions, the same pushing/call/return/popping operation will occur.
- The stack keeps track of the order and depth of the function calls.

2.4 Implementing the Stack

- To implement the stack in a CPU, we need a few things:
    1. A **stack pointer**, which points at the top-most item on the stack.

2. Instructions to implement pushing and popping.
3. A large amount of RAM which can be used to hold the stack's contents.

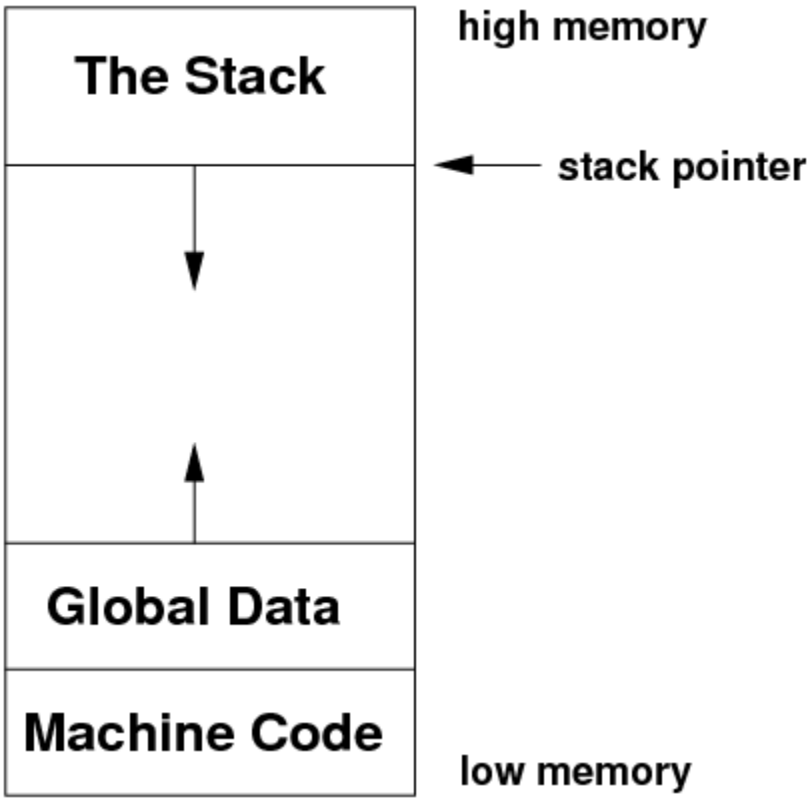
- Implementing the stack pointer is easy if our CPU provides indirect addressing: we can reserve one register to be the stack pointer, or we can reserve a specific memory address to be the stack pointer.
- On the MIPS ISA, we already know enough instructions to implement pushing and popping. For example, to push a value:

```
subu $sp, $sp, 4     # Decrement $sp by 4
sw $t1, ($sp)        # and copy $t1 onto the stack
```

- Popping a data item is the reverse:

```
lw $t1, ($sp)
addu $sp, $sp, 4
```

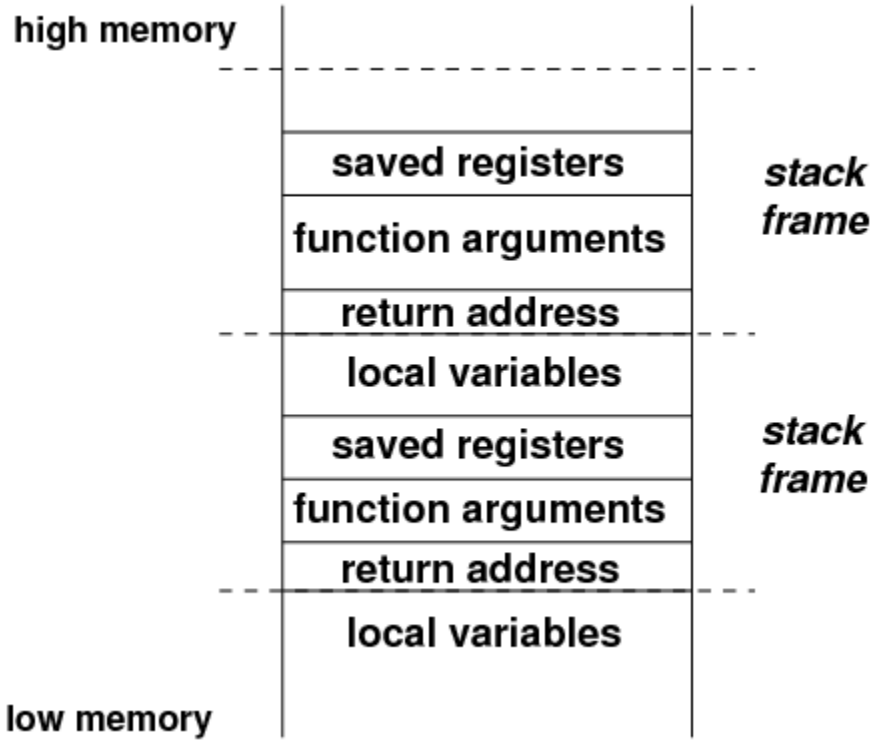
- On some CPUs (usually the CISC ones), there are individual PUSH and POP instructions.
- Where to place the stack is another question? It needs to go in main memory somewhere, but given that it grows, where to put it?
- One common solution is to place it at the top of memory, growing downwards (i.e. an upside-down stack):



- The program's machine code is fixed in size, and can be placed at the bottom. Above that is the program's global data, and the arrangement allows the program's global data to grow as well.

2.5 The Stack Frame

- We have PUSH and POP operations on the stack, but these are for individual items.
- When dealing with functions, there are going to be a lot of items pushed onto the stack.
- We call the group of items related to a function call a **stack frame**.
- A typical set of stack frames looks like:



- Each frame has been drawn with dotted lines to indicate whose responsibility it is to remove the frame from the stack, i.e. the function caller.
    - In this format, the function caller owns the arguments that it sends to the function.
    - In some texts, the frame is drawn with the arguments belonging to the function, not the function caller.
- Each frame holds:
    - any registers saved
    - the arguments to the function being called
    - the return address
- But also note that the frame has the function's own local variables.
- The stack is the natural place to store local variables. Because these are not at fixed or named locations in memory, other functions will not know where they are, which keeps them somewhat private.

- And the stack can grow to any size, so each function can create as many local variables as it needs, as long as it remembers to remove them from the stack before it returns.

2.6 Returning Values from a Function Call

- Mathematically speaking, functions return a single value.
- In computing, we do not need to be limited by this. On computers, we can return zero, one, or any number of results.
- If there are only a few return values, the simplest way to return them is to use registers.
- For large items, it makes sense to store them in memory somewhere, and return pointers to the data items (i.e. in registers).
- Beware though! Careful not to make the mistake of returning a pointer to a local variable. Why not?
- Local variables are kept on the stack. As a function, if you return a pointer to one of your own local variables, then there is a good chance that it will be overwritten on the next function call.
- If the function caller expects to be able to use your data, then it will be corrupted after the next function call made by the caller.

3 Function Calls on MIPS

- Let's now look at the ISA provided instructions available to call functions, return from functions, deal with the stack and the stack frames.
- As you would expect, on the RISC MIPS CPU, support is minimal.
- One register is reserved as the stack pointer, $sp, and the return address for each function is stored in the $ra register.
- This immediately implies that each function has to manually store the $ra on the stack before they make a function call, otherwise their own return address (in $ra) will be destroyed by the function call.
- Two instructions for function calls exist: JAL (jump and link i.e. non leaf procedures) and JR (jump register i.e. leaf procedures).

    JAL address

    $ra <- address of the instruction after the JAL, so the function can return here

    PC <- address in the JAL instruction, i.e. jump to the start of function

    JR $nn

    PC <- address stored in the $nn register

- That is it, there is no ISA support for the stack frame, so it all has to be managed manually.
- There is a convention for the format of the MIPS stack frame, and who manages which part; we will look at this in the lab.

====

3.1 Instructions for Calling and Returning

MIPS provides two instructions for calling functions and returning from functions:

1. `jal label`: Jump to the instruction at the named label, storing the address of the instruction following the jal in the $ra (return address) register.
2. `jr $ra`: Jump back to the instruction at the address stored in the $ra (return address) register.

These are the only instructions we need to do function calls; however, things are not as simple as they seem.

3.1 Task 1

Download the program **w6jal_jr_instructions.asm** and load it into MARS. This program starts in the *main()* program, which prints out one string. *main()* then calls the *function()*using the *jal* instruction, which prints out the second string. Once the *function()* returns using the *jr* instruction, the *main()* program prints out the first string again.

Assemble the program, and slowly single-step the program. Watch the first string get printed out with $a0 pointing at the first string. When the *jal* instruction is performed, notice the program jump to the *function(),* and also notice that $ra is set pointing at the instruction after the *jal*. Watch the second string bring printed. When the *jr* instruction is performed, see the program jump back to the *main()* program.

Question: why did the *main()* program print out the second string and not the first string again? After all, the only value that the *main()* program puts into $a0 is the address of string1.

The answer is that the *function()* overwrote the value in $a0 when it printed out the second string, so when we returned back to *main(),* what *main()* thinks is in the $a0 register is no longer there. The *function()* has tromped all over *main()*'s register value!

4 Functions: What We Need

Some of the following is abstracted from Appendix A of Patterson and Hennessy's book.

You can see that, if we are going to call and return from functions properly, we need to do some things other than jal & jr to get it to work. Here are the things we need for functions to work:

1. The function caller needs to be able to provide a set of arguments, in a specific order, to the function. The function needs to be able to see these as local variables.
2. The function needs to be able to create its own local variables which are different from those in the functions which called it.
3. The function caller and the function itself are going to need to agree on how to store away register values, so that each will have access to the CPU's registers without having to worry about losing values.
4. The function needs to be able to return a result to the function caller.
5. We also want recursion: the ability to write functions which call themselves an arbitrary number of times.

In the following discussion, the function being called is going to be known as the *callee*.

4.1 Registers for Arguments and Result

On a CPU, there are two places that we could use to send in arguments to a function, and get back the result: registers or main memory. With the MIPS CPU, the registers $a0 to $a3 are used to send in up to 4 arguments to a function: if the function needs more, then the caller and callee must write to store and collect the remaining arguments in main memory. The registers $v0 and $v1 are used to return a single result to the caller: both are used in combination if the result is a doubleword.

4.2 Who Deals With Which Registers?

Of course, there are only 32 registers on the MIPS CPU, and both the function caller and the function callee will need to use some of them. *Somebody* is going to need to do some work in order to save the function caller's existing values, so that the function itself has some registers to use.

Here is the register saving convention used by all software written for the MIPS:

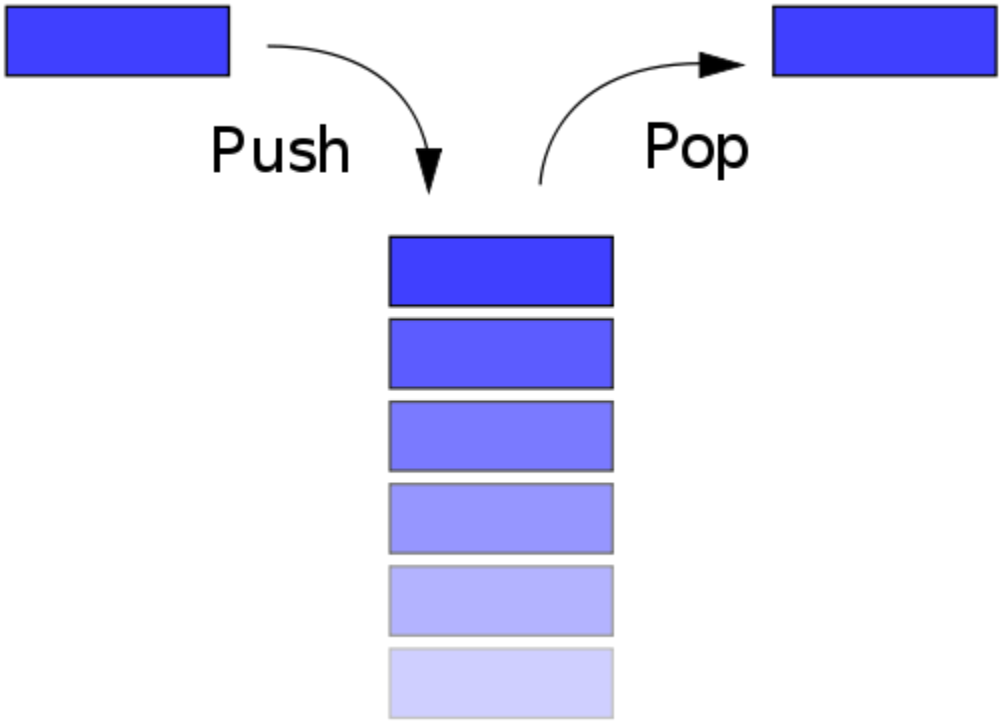
- Registers $a0 to $a3 are used for arguments, and registers $v0 and $v1 for the return value. Similarly, the caller may also be using the temporary registers $t0 to $t7. These are all *caller-saved* registers: if the caller expects to use one of these registers after a call, it must save its value before the call.
- A callee must save the values in these registers ($s0 ... $s7, $fp, and $ra) before altering them since the caller expects to find these registers unchanged after the call. These are *callee-saved* registers.

4.3 The Stack

So, the function caller stashes some registers away before it makes the function call. Likewise, the function callee stashes some registers away when it gets called. On the way back out, both the callee and caller restore the register values, so that when the function is done the caller has its registers back the way they were (except for the result of the function).

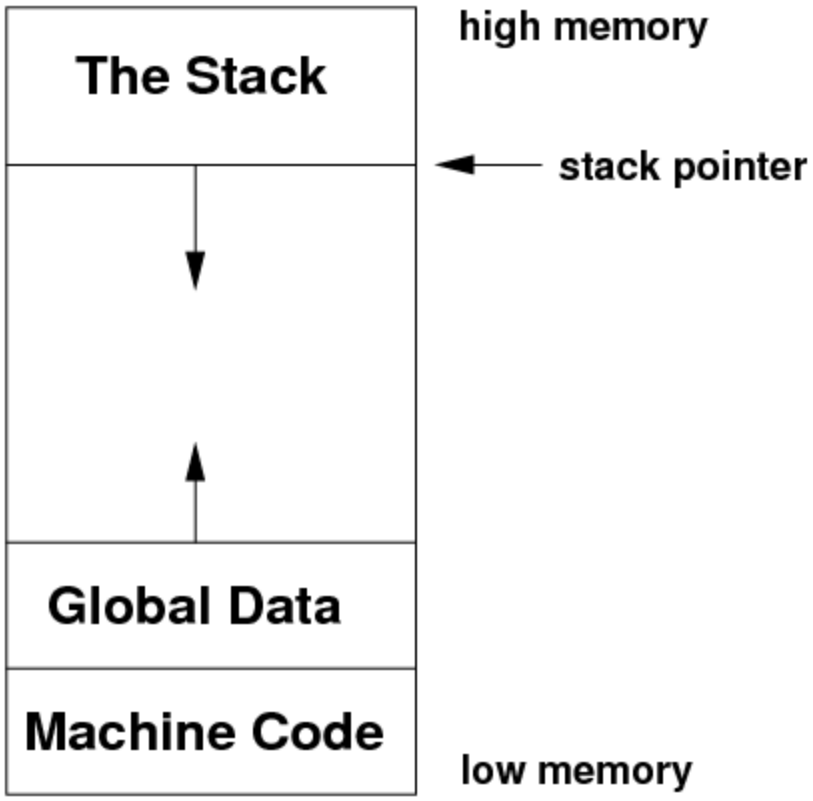
This is all fine and good if we only call one function at a time: we can choose a fixed set of memory locations and store the register values into these locations, then fetch them back later. But what about functions that call functions, and functions that perform recursion an arbitrary number of times? One fixed set of locations is not going to be enough.

This issue is solved using a *stack*. A stack is an area of memory of (essentially) arbitrary size which we can use to store as many register values as we like, as well as other data. We can *push* values onto the stack, which grows in size, and later we can *pop* values back off the stack in reverse order.



Stack (from Wikipedia)

Conventionally, memory on many computers is laid out so that the code of a program is stored in low memory, followed by any global data which has been allocated to the program. The stack is created at the high-end of memory and as it grows, it grows downwards in memory locations. In other words, computer stacks are upside down: a push makes it grow at the bottom, not the top!



This division of a program's memory into three distinct areas is not the only way to do things, but it does allow both the global data and the stack (where local data is stored) to expand with minimum risk of running into each other.

Note the existence of the *stack pointer*, which points at the item most recently pushed onto the stack, and the item which is the first to be pulled. The MIPS register $sp is used as the stack pointer. The memory at addresses below the stack pointer is not yet allocated, but will become used once new data items are pushed onto the stack. When a program begins execution, the operating system will initialize the stack pointer to the highest address which is part of the stack.

4.4 Pushing and Popping

Pushing a data item onto the stack is easy: decrement $sp by the size of the item, and store the item on the stack using $sp indirect addressing. For example, to push the value in $t1 onto the stack:

```
subu $sp, $sp, 4      # Decrement $sp by 4
sw $t1, ($sp)         # and copy $t1 onto the stack
```

Popping a data item is the reverse. Let's pop the top word off the stack and bring it back into the $t3 register:

```
lw $t3, ($sp)
addu $sp, $sp, 4
```

Note: memory addresses start at 0 and go higher: there are no negative addresses. This explains why the *addu* and *subu*instructions are used.

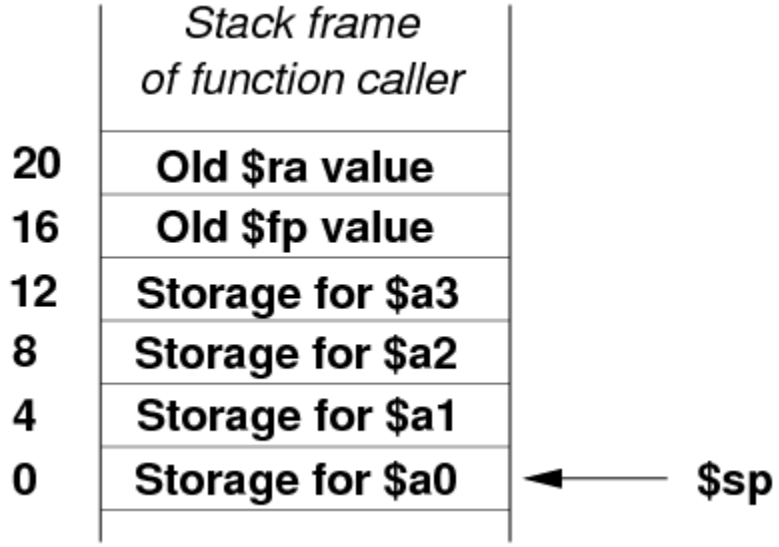
4.5 Stack Frames

Even though our discussion has shown you how to push and pop individual data items, in fact we will be growing and shrinking the stack in whole groups of data items. We know that each function has to keep track of:

- the arguments passed to it from its caller
- the address to return to when the function returns
- any callee-saved registers that it saved when the function started up
- any local variables which are accessible only to this function
- when calling another function:
    - any caller-saved registers that it must save before making the call
    - any arguments to the function that it is calling

To do this, each function builds a *stack frame* on the stack which records all of these details. The stack frame has a generic format, although each frame's size will vary depending on how many arguments, saved registers and local variables are stored on it.

The simplest and smallest stack frame is shown in the diagram below; it is 24 bytes in size.



The first thing we (as a function) must do when it is called is to move the stack pointer $sp down to make room on the stack for the frame. The next critical thing we need to do, then, is to save the return address currently in $ra. Why? Because if we ourselves now call another function, the return address that we need will be lost!

At the top of each function, therefore, should be these instructions or similar:

```
function:  subu $sp, $sp, 24      # Make a stack frame of 24 bytes
           sw $ra, 20($sp)        # Save our $ra before we call another
                                  # function
                                  # Note: we treat the stack frame like
                                  # an array
```

And the last instructions in a function should undo the above two instructions:

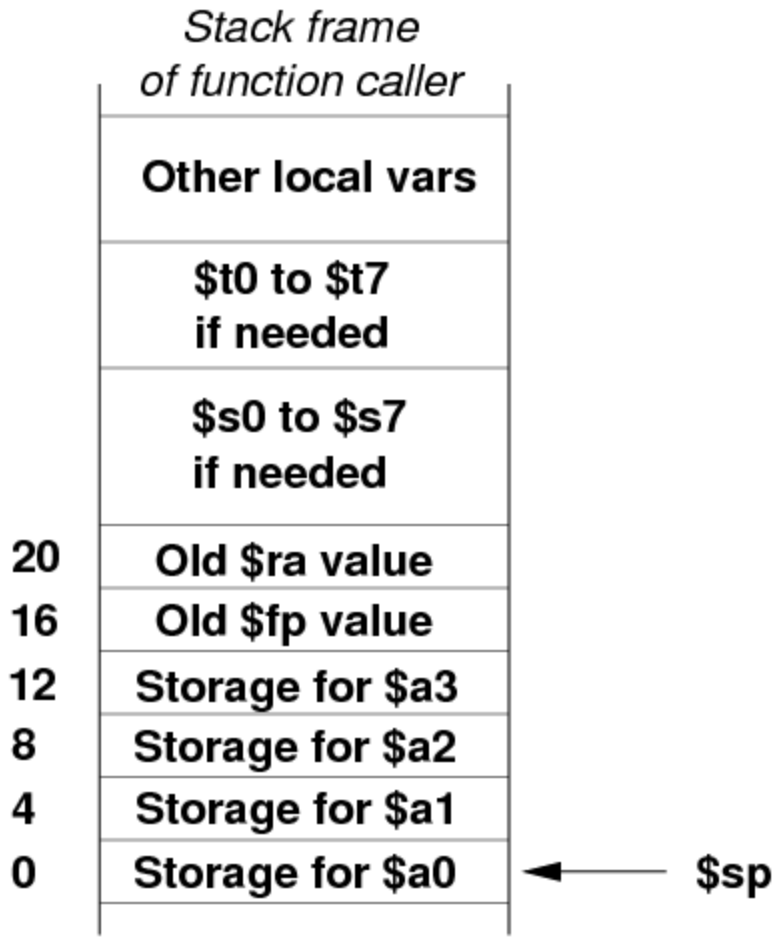
```
           lw $ra, 20(sp)         # Reload our return address
           addu $sp, $sp, 24      # Destroy our stack frame
           jr $ra                 # and return out of this function
```

4.6 What About the Rest of the Stack Frame?

Good question! The minimal stack frame provides room for us, the recently-called function, to stash away copies of our arguments in $a0 to $a3, before we ourselves call a function. However, saving these registers is not mandatory: we only need to do it if we are calling another function. Similarly, there is room to save our $fp register before we call a function which might destroy it.

Now, the stack frame shown above is simply the minimal, smallest stack frame. If we need to save other registers on the stack, then we will create a larger stack frame, and we get to choose where to put the extra values on the stack frame.

I have tried to find a definitive guide for what an extended MIPS stack frame should look like, but there does not seem to be any general agreement. So I recommend the following stack frame structure for this subject:



Obviously, you need to design your stack frame to hold those values that will get overwritten, but you don't need to make the stack frame any bigger than that.

Aside Note: To simplify for our course discussion on this subject we may not be using the $fp register at all in our examples, so there is no need for any function to save a copy of it on the stack frame.

5 Local Variables

Functions also want to keep variables which are local, i.e. not globally visible and only within the scope of the function while it is running. If we used `.data` to store these variables, then the assembler would allocate space for them in the global data area, and everybody would be able to see them.

Instead, we can use the stack frame to store local variables. We need to:

1. increase the size of the stack frame to provide enough space to hold the local variables.
2. decide where in this extended stack frame to place our local variables.

6 Recursion

We can now finally tackle recursion. For recursion to work, a function has to be able to call itself, and it also has to be able to keep its own local variables which will not be clobbered by other instances of the function.

For the local variables, we have to consider: caller-save registers, callee-save registers, storing locals in the stack frame, and/or all of the above.

Consider the prototypical recursive function, *fact(X)*:

```
int fact(int X) {
  int result;

  if (X==1) return(1);
  result= X * fact(X-1);
  return(X);
}
```

Here, the *X* variable is both an argument to the function and a local variable in the function. Similarly, *result* is a local variable in the function. We don't have to worry too much about the value in *result*, as it is unknown before the recursive function call, and filled in by the recursive function call. However, the value in *X* has to be preserved across the function call, so we can multiply the result by *X* after the function call.

6.1 Task 3

Download the program **w6factorial.asm** and load it into MARS. This program implements the recursive factorial function in MIPS assembly. Our argument (shown as *X* above) comes in as the $a0 register. Therefore, we need to preserve $a0 across the recursive call to the function. This is done by the `sw $a0, 0($sp)` instruction before the `jal` function call, and the `lw $a0, 0,($sp)` instruction after the return from the function.

7 Your turn…Program to debug and test.

7.1 Task 1

Write a program that does the same job as 3.1 Task 1 (w6jal_jr_instructions.asm), but this time both the *main()* program and the *function()*create stack frames for themselves. This allows *main()* to keep the value in $a0 across the call to the *function()*.

Run, and single-step, the program and make sure that you understand how the stack frame is working.

8 Outlook for the Next Lab

In the next lab, we will look at Computer Arithmetic including exceptions, interrupts and exception handling on the MIPS CPU.