# MIPS Calling Convention

This document summarizes the calling conventions that we expect you to use in ECE 314 for homework problems and assembly language programming projects. These are the rules for how registers should be used and how stack frames should be laid out in memory.

Unfortunately, there is actually no such thing as "The MIPS Calling Convention". Many possible conventions are used by many different programmers and assemblers. Several different approaches are outlined in the text, but none is fully described nor documented.

## Register Use

For convenience, we repeat the register usage conventions here:

| Number | Name | Purpose |
|---|---|---|
| $0 | $0 | Always 0 |
| $1 | $at | The *Assembler Temporary* used by the assembler in expanding pseudo-ops. |
| $2-$3 | $v0-$v1 | These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant. |
| $4-$7 | $a0-$a3 | The *Argument* registers, these registers contain the first 4 argument values for a subroutine call. |
| $8-$15,$24,$25 | $t0-$t9 | The *Temporary Registers*. |
| $16-$23 | $s0-$s7 | The *Saved Registers*. |
| $26-$27 | $k0-$k1 | The *Kernel Reserved registers*. DO NOT USE. |
| $28 | $gp | The *Globals Pointer* used for addressing static global variables. For now, ignore this. |
| $29 | $sp | The *Stack Pointer*. |
| $30 | $fp (or $s8) | The *Frame Pointer*, if needed (this was discussed briefly in lecture). Programs that do not use an explicit frame pointer (e.g., everything assigned in ECE314) can use register $30 as another saved register. Not recommended however. |
| $31 | $ra | The *Return Address* in a subroutine call. |

The shaded rows indicate registers whose value must be preserved across subroutine calls.

## Stack Frames

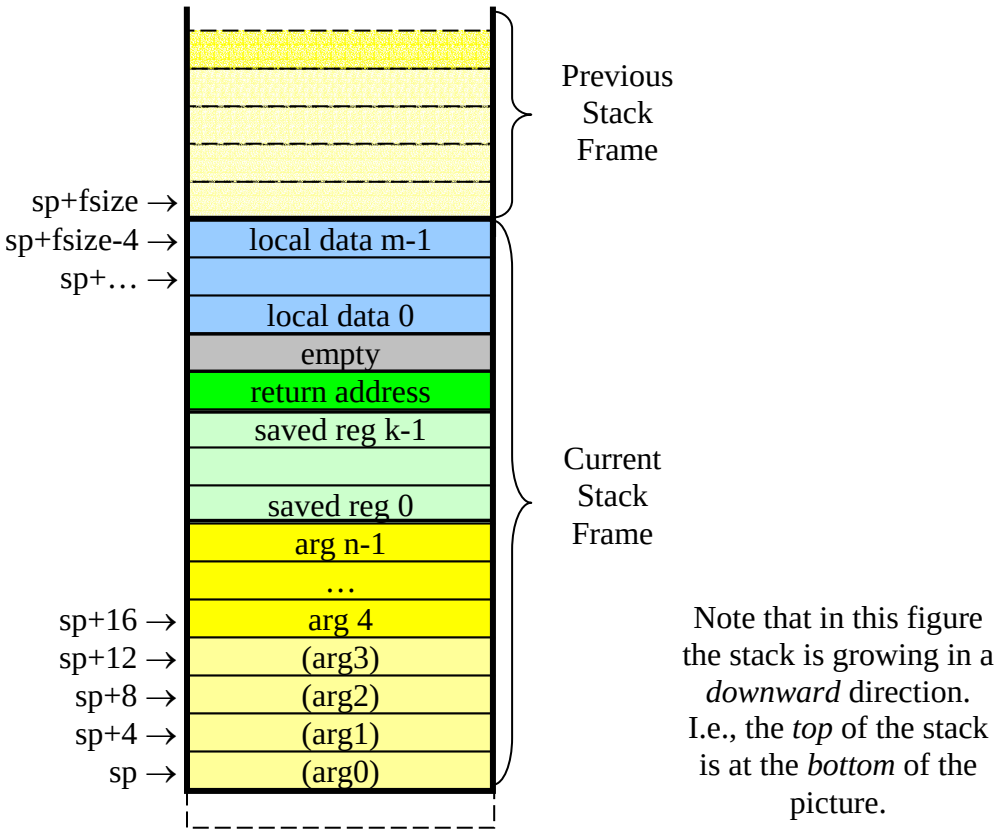
Remember, every time a subroutine is called, a unique stack frame is created for that instance of the subroutine call. (In the case of a recursive subroutine call, multiple stack frames are created, one for each instance of the call.) The organization of the stack frame is important for two reasons. One, it forms a contract between the caller and the callee that defines how arguments are passed between the caller and callee, how the result of a function is passed from the callee to the caller, and defines how registers are shared

between the caller and the callee. Second, it defines the organization of storage local to the callee within its stack frame.

In general, the stack frame for a subroutine may contain space to contain the following:

- Space to store arguments passed to subroutines that this subroutine calls.
- A place to store the values of saved registers ($s0 to $s7).
- A place to store the subroutine's return address ($ra).
- A place for local data storage.

The following figure illustrates the general organization of a stack frame.




We divide the stack frame into five regions

1. The Argument Section of a stack frame contains the space to store the arguments that are passed to any subroutine that are called by the current subroutine (i.e., the subroutine whose stack frame currently on top of the stack.) The first four words of this section are never used by the current subroutine; they are reserved for use by any subroutine called by the current subroutine. (Recall that up to four arguments can be passed to a subroutine in the argument registers ($a0 to $a3). If

there are more than four arguments, the current subroutine stores them on the stack, at addresses sp+16, sp+20, sp+24, etc.

2. The Saved Registers Section of the stack frame contains space to save the values of any of the saved registers ($s0 to $s7) that the current subroutine wants to use. On entry into the current subroutine, the subroutine copies the values of any of the saved registers, $s0 to $s7, whose values it might change during the course of the execution of the subroutine into this section of the stack frame. Just before the subroutine returns, it copies these values from the Saved Registers Section back into the original saved registers. In between, the current subroutine is free to change the value of the saved registers at will. However, the caller of the current subroutine will see the same values in these registers after the subroutine call as it saw before the subroutine call.
3. The Return Address Section is used to store the value of the return address register, $ra. This value is copied onto the stack at the start of execution of the current subroutine and copied back into the $ra register just before the current subroutine returns.
4. The Pad is inserted into the stack frame to make sure that total size of the stack frame is always a multiple of 8. It is inserted here to ensure that the local data storage area starts on a double word boundary.
5. The Local Data Storage Section is used for local variable storage. The current subroutine must reserve enough words of storage in this area for all of its local data, including space to store the value of any temporary registers ($t0 to $t9) that it needs to preserve across subroutine calls. The local data storage area must also be padded so that its size is always a multiple of 8 words.

A couple of additional rules that do not obviously follow from the above:

- The value of the stack pointer is required to be multiple of 8 at all times. This ensures that a 64-bit data object can be pushed on the stack without generating an address alignment error at run-time. This implies the size of every stack frame must be a multiple of 8; technically, this requirement applies even to leaf subroutines as discussed below.
- The values of the argument registers $a0-$a3 are not required to be preserved across subroutine calls. Thus, a subroutine is allowed to change the values of any of the argument registers without saving/restoring them.
- The first four words of the Argument Section of a stack frame are known as *argument slots* -- memory locations reserved to store the four arguments $a0-$a3. It is important to remember that a subroutine does *not* store anything in the first four argument slots, the actual arguments are passed in $a0 to $a3. However, the called subroutine may choose to copy the values of $a0 to $a3 into the argument slots if it wants to save these values. If it does so, then it can then treat all its arguments as a 1-dimensional array in memory.
- There are several very important things to note about the argument slots.
    - The argument slots are allocated by the caller but are used by the callee!

- All four slots are *required*, even if the caller knows it is passing fewer than four arguments. Thus, on entry a subroutine may legally store *all* of the argument registers into the argument slots if desired.
- The caller must allocate space on its stack frame for the maximum number of arguments for *any* subroutine that it calls (or the minimum of four arguments, if all the subroutines that it calls have fewer than four arguments.)

**Leaf vs Nonleaf Subroutines**

Not every subroutine will need every section described above in its stack frame. The general rule is that if the subroutine does not need a section, then it may *omit* that section from its call frame. To help make this clear we will distinguish 3 different classes of subroutines:

- *Simple Leaf* subroutines do not call any other subroutines, do not use any memory space on the stack (either for local variables or to save the values of saved registers). Such subroutines do not require a stack frame, consequently never need to change $sp.
- *Leaf with Data* are leaf subroutines (i.e. do not call any other subroutines) that require stack space, either for local variables or to save the values of saved registers. Such subroutines push a stack frame (the size of which should be a multiple of 8 as discussed above). However, $ra is not saved in the stack frame.
- *Nonleaf* subroutines are those that call other subroutines. The stack frame of a nonleaf subroutine will probably have most if not all the sections.

Below are examples for each of these cases.

**A Simple Leaf Function**

Consider the simple function

```
int g( int x, int y ) {
  return (x + y);
}
```

This function does not call any other function, so it does not need to save $ra. Also, it does not require any temporary storage. Thus, it can be written with no stack manipulation at all. Here it is:

```
g:      add   $v0,$a0,$a1 # result is sum of args
        jr    $ra # return
```

Because it has no local data, this function does no stack manipulation at all; its stack frame is of zero size.

**A Leaf Function With Data**

Now let's make g a little more complicated:

```
int g( int x, int y ) {
  int a[32];
  ... (calculate using x, y, a);
  return a[0];
}
```

This function does not call any other function, so it does not need to save $ra, but, it does require space for the array a. Thus, it must push a stack frame. Here is the code:

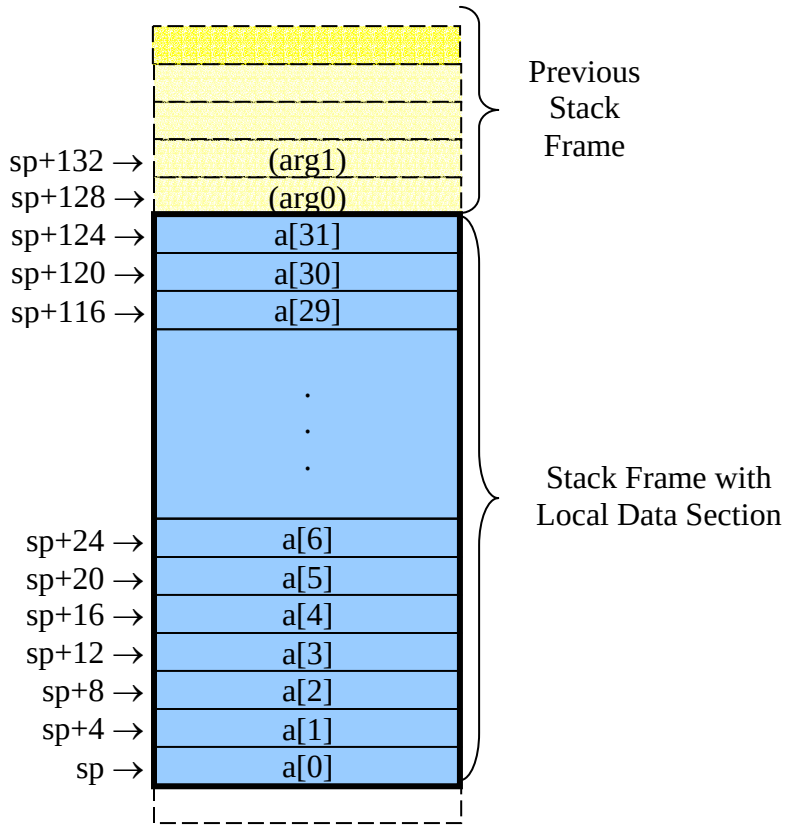
```
g:      # start of prologue
        addiu $sp,$sp,(-128) # push stack frame
        # end of prologue

        . . .                # calculate using $a0, $a1 and a
                             # array a is stored at addresses
                             # 0($sp) to 124($sp)

        lw    $v0,0($sp)     # result is a[0]

        # start of epilogue
        addiu $sp,$sp,128    # pop stack frame
        # end of epilogue

        jr    $ra            # return
```

Because this is a leaf function, there is no need to save/restore $ra and no need to leave space for argument slots. This stack frame only needs space for local data storage. Its size is 32 words or 128 bytes. Also, we always require that the local data storage block be double word aligned, but in this case since the value of $sp is always double word aligned, no padding is necessary. The array a is stored at addresses 0($sp) to 124($sp).

Also we have gathered the instructions necessary to set up the stack frame into a *prologue* and the instructions necessary to dismantle the stack frame into an *epilogue* to keep them separate from the code for the body of the subroutine.

Note that, in this example, the code for the calculations (…) is *not* allowed to change the values of any of the registers $s0 to $s7.

The stack frame for this example follows.




**A Leaf Function With Data and Saved Registers**

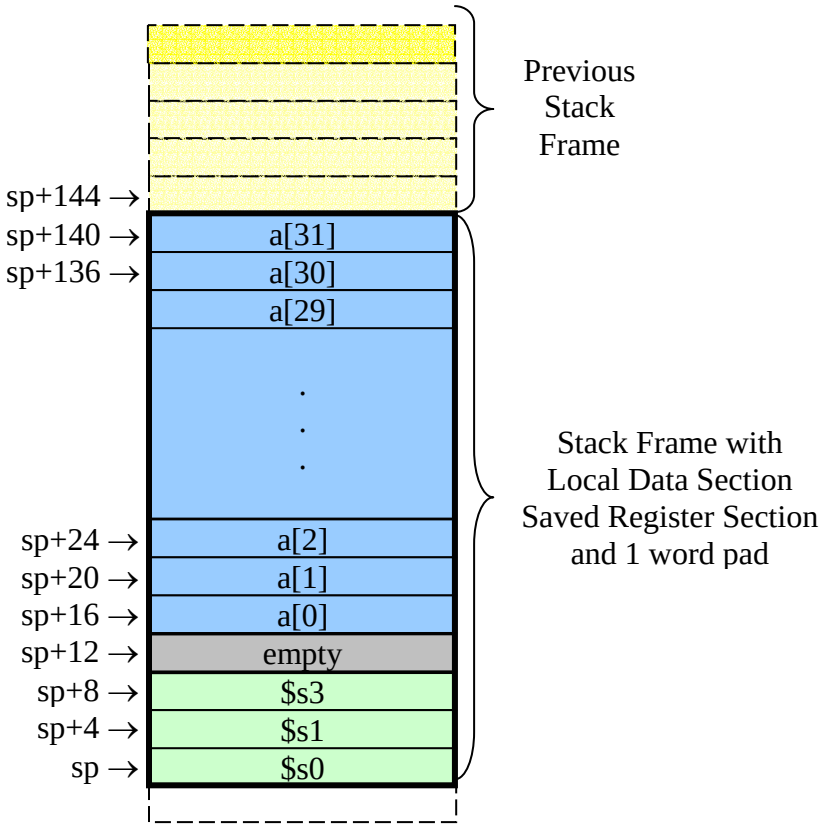
Suppose that the code for the calculations (…) *does* change the value of some of the saved registers, $s0, $s1, and $s3. We would have to change code for g as follows.

```
g:      # start of prologue
        addiu $sp,$sp,(-144) # push stack frame

        sw $s0,0($sp)        # save value of $s0
        sw $s1,4($sp)        # save value of $s1
        sw $s3,8($sp)        # save value of $s3
        # end of prologue

        # start of body
        . . .                # calculate using $a0, $a1 and a
                             # array a is stored at addresses
                             # 16($sp) to 140($sp)

        lw $v0,16($sp)       # result is a[0]
        # end of body

        # start of epilogue
        lw $s0,0($sp)        # restore value of $s0
        lw $s1,4($sp)        # restore value of $s1
        lw $s3,8($sp)        # restore value of $s3
```

```
        addiu $sp,$sp,144    # pop stack frame
        # end of epilogue

        jr    $ra            # return
```

In this case, we had to add a Saved Register Section and Pad to the stack frame. The Saved Register Section consists of three words, at addresses 0(sp), 4(sp), and 8(sp), that are used to save the value of $s0, $s1, and $s3. The Pad, at address 12(sp) is used to pad the stack frame so that its size is a multiple of 8 and start to the Local Data Section at an address that is also a multiple of 8. The total size of this stack frame is 36 (3+1+32) words or 144 (12+4+128) bytes. Instructions to save the saved registers have been added to the prologue and instructions to restore the saved registers have been added to the epilogue. Note that the addresses used to store the array a increased by 16 as a result of adding this new section, so the array is now stored at addresses 16($sp) to 140($sp).

The stack frame for this example follows.



**A Nonleaf Subroutine**

Now let's make g even more complicated:

```
int g( int x, int y ) {
  int a[32];

  ... (calculate using x, y, a);

  a[1] = f(y,x,a[2]);
  a[0] = f(x,y,a[1]);
  return a[0];
}
```

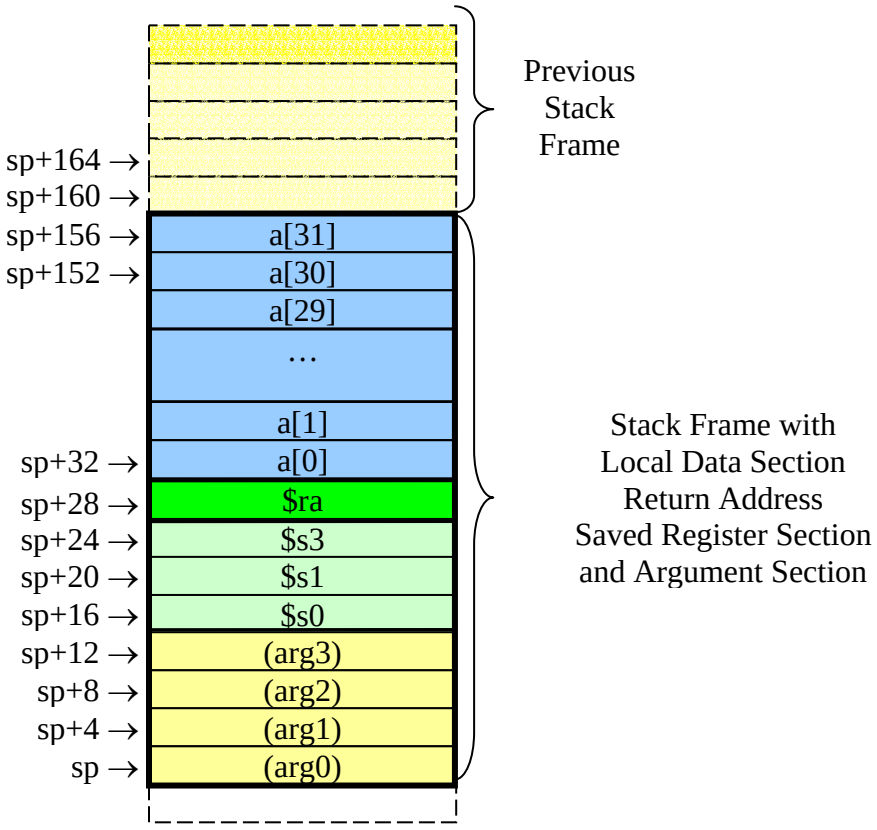
Now g makes two calls to a function f. We would have to change the code for g as follows.

```
g:
        # start of prologue
        addiu $sp,$sp,(-160) # push stack frame

        sw $ra,28($sp)       # save the return address

        sw $s0,16($sp)       # save value of $s0
        sw $s1,20($sp)       # save value of $s1
        sw $s3,24($sp)       # save value of $s3
        # end of prologue

        # start of body
        . . .                # calculate using $a0, $a1 and a
                             # array a is stored at addresses
                             # 32($sp) to 156($sp)

        # save $a0 and $a1 in caller's stack frame
        sw $a0,160(sp)       # save $a0 (variable x)
        sw $a1,164(sp)       # save $a1 (variable y)

        # first call to function f
        lw $a0,164(sp)       # arg0 is variable y
        lw $a1,160(sp)       # arg1 is variable x
        lw $a2,40(sp)        # arg2 is a[2]
        jal f                # call f

        sw $v0,36(sp)         # store value of f into a[1]

        # second call to function f
        lw $a0,160(sp)       # arg0 is variable x
        lw $a1,164(sp)       # arg1 is variable y
        lw $a2,36(sp)        # arg2 is a[1]
        jal f                # call f

        sw $v0,32(sp)        # store value of f into a[0]

        # load return value of g into $v0
        lw $v0,32($sp)       # result is a[0]
```

```
# end of body

# start of epilogue
lw $s0,16($sp)       # restore value of $s0
lw $s1,20($sp)       # restore value of $s1
lw $s3,24($sp)       # restore value of $s3

lw $ra,28($sp)       # restore the return address

addiu $sp,$sp,160    # pop stack frame
# end of epilogue

jr    $ra            # return
```

In this example, we have added the Argument Section and the Return Address Section to the stack frame and omitted the Pad. Since f, the function that g calls, has 3 arguments, we add the minimum size Argument Section of 4 words. The total size of this stack frame is 40 (4+3+1+32) words or 160 (16+12+4+128) bytes. Once again the addresses used to store the array a increased as a result of adding the new sections, so the array is now stored at addresses 32($sp) to 156($sp).

The stack frame for this example follows.



**Another Nonleaf Subroutine**

Now let's make g even more complicated:

```
int g( int x, int y ) {
  int a[32];

  ... (calculate using x, y, a);

  a[1] = f(y,x,a[2],a[3],a[4]);
  a[0] = f(x,y,a[1],a[2],a[3]);
  return a[0];
}
```

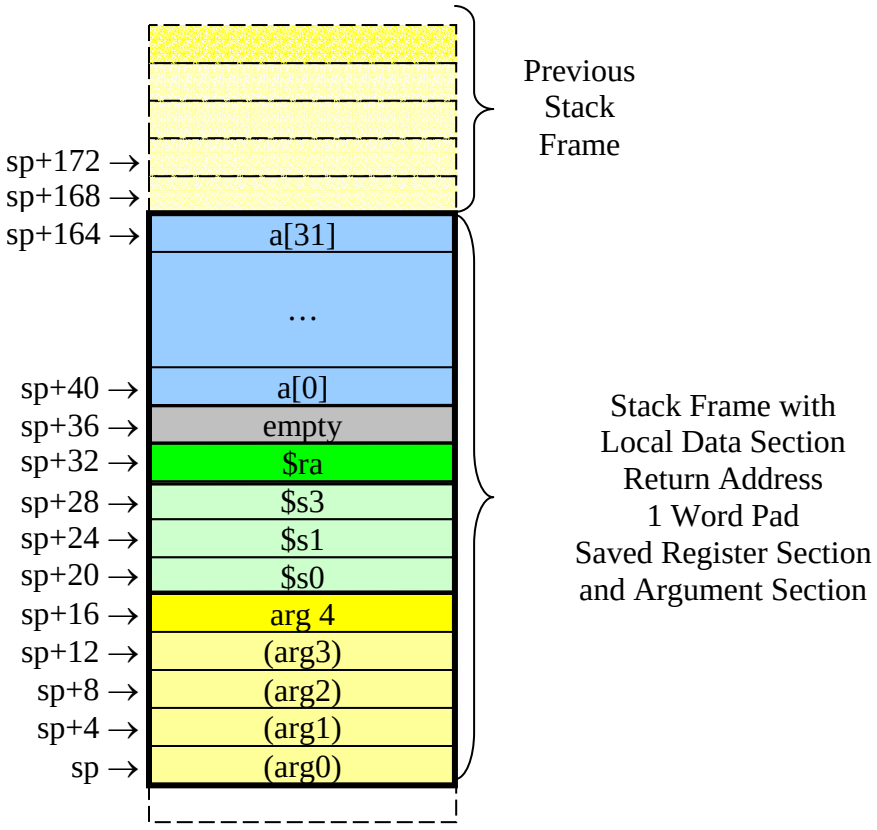
Now g makes two calls to a function f, but now f has five arguments. We would have to change the code for g as follows.

```
g:
        # start of prologue
        addiu $sp,$sp,(-168) # push stack frame

        sw $ra,32($sp)       # save the return address

        sw $s0,20($sp)       # save value of $s0
        sw $s1,24($sp)       # save value of $s1
        sw $s3,28($sp)       # save value of $s3
        # end of prologue

        # start of body
        . . .                # calculate using $a0, $a1 and a
                             # array a is stored at addresses
                             # 40($sp) to 164($sp)

        # save $a0 and $a1 in caller's stack frame
        sw $a0,168(sp)       # save $a0 (variable x)
        sw $a1,172(sp)       # save $a1 (variable y)

        # first call to function f
        lw $a0,172(sp)       # arg0 is variable y
        lw $a1,168(sp)       # arg1 is variable x
        lw $a2,48(sp)        # arg2 is a[2]
        lw $a3,52(sp)        # arg3 is a[3]
        lw $t0,56(sp)        # arg4 is a[4]
        sw $t0,16(sp)        # BUT it is passed on the stack!
        jal f                # call f

        sw $v0,44(sp)         # store value of f into a[1]

        # second call to function f
        lw $a0,168(sp)       # arg0 is variable x
        lw $a1,172(sp)       # arg1 is variable y
        lw $a2,44(sp)        # arg2 is a[1]
        lw $a3,48(sp)        # arg3 is a[2]
        lw $t0,52(sp)        # arg4 is a[3]
        sw $t0,16(sp)        # BUT it is passed on the stack!
```

```
jal f                 # call f

sw $v0,40(sp)         # store value of f into a[0]

# load return value of g into $v0
lw $v0,40($sp)        # result is a[0]
# end of body

# start of epilogue
lw $s0,20($sp)        # restore value of $s0
lw $s1,24($sp)        # restore value of $s1
lw $s3,28($sp)        # restore value of $s3

lw $ra,32($sp)        # restore the return address

addiu $sp,$sp,168     # pop stack frame
# end of epilogue

jr    $ra             # return
```

In this example, since f, the function that g calls, has 5 arguments, we have increased the size of the Argument Section to five words and added a Pad. The total size of this stack frame is 42 (5+3+1+1+32) words or 168 (20+12+4+4+128) bytes. Now the addresses used to store the array a start at 40(sp) and end at 164($sp).

The stack frame for this example follows.



**Summary**

The following summarizes the ECE314 MIPS calling convention. A stack frame has 0 to 5 sections depending on the requirements of the subroutine.

A stack frame has a **Argument Section,**

- If the subroutine calls other subroutines (i.e., is a non-leaf subroutine).
- Don't forget that

1. The Argument Section is large enough to hold the largest number of arguments of any subroutine called by the current subroutine. (One word is reserved per argument.)
2. The Argument Section has a minimum size of 4 words.
3. The argument registers $a0, $a1, $a2, and $a3 are used to pass the first four arguments to a subroutine. These arguments are not copied into the Argument Section by the calling subroutine, although space is reserved for them. Additional arguments are copied to the Argument Section (at offsets 16, 20, 24, etc.)
4. A subroutine that is called by the current subroutine, may, but is not required to, copy the contents of the argument registers into the Argument Section of the current subroutine (at offsets 0, 4, 8 and 12.)
5. The Argument Section is found, at the low end of the stack frame.

A stack frame has a **Saved Registers Section,**

- If the subroutine want to use (i.e., change the value) of any of the saved registers, $s0 to $s7.
- Don't forget that

1. The Saved Registers Section contains 1 word for each register that it is required to save on entry and restore on exit.
2. The subroutine prologue must copy the value of each register that whose value it must save into the Saved Registers Section on entry into the subroutine (e.g., in the subroutine prologue) and must copy the value of each of these registers back into the correct register before it returns (e.g., in the subroutine epilogue.)
3. The subroutine is not required to save/restore the value of a saved register if it does not change the value of the register in the subroutine body.
4. The Saved Registers Section is found just above the Argument Section.

A stack frame has a **Return Address Section,**

- If the subroutine calls other subroutines (i.e., is a non-leaf subroutine).
- Don't forget that

1. The Return Address Section contains exactly 1 word.
2. The value of the return address register, $ra, is copied to the Return Address Section on entry into the subroutine (e.g., in the subroutine prologue) and copied from the Return Address Section before exit from the subroutine (e.g., in the subroutine epilogue.)
3. The Return Address Section is found just above the Saved Registers Section.

A stack frame has a **Pad,**

- If the number of bytes in the Argument Section, Saved Registers Section, and the Return Address Section is not a multiple of 8.
- Don't forget that

  1. The Pad contains exactly 1 word.
  2. Nothing is stored in the Pad.
  3. The Pad is found just above the Return Address Section.

A stack frame has a **Local Data Storage Section,**

- If the subroutine has local variables or must save the value of a register that is not preserved across subroutine calls.
- Don't forget that

  1. The size of the Local Data Storage Section is determined by the local data storage requirements of the subroutine, but must always be a multiple of 8 bytes.
  2. The internal organization is entirely determined by the subroutine. This part of the stack is private to the subroutine and is never accessed by any other subroutine, either a caller or callee of the current subroutine.
  3. The Local Data Storage Section is found just above the Pad. This places it at the high end of the stack frame.

The total size of the stack frame for a subroutine is the sum of the sizes of the individual sections (using 0 for the size of sections that it doesn't need.) The subroutine should push its stack frame onto the stack on entry into the subroutine by subtracting the size of its stack frame from the stack pointer, $sp. The subroutine should pop its stack frame off from the stack just before exit from the subroutine by adding the size of its stack frame to the stack pointer, $sp. Make sure that you add the same amount at the end as you subtracted at the beginning!