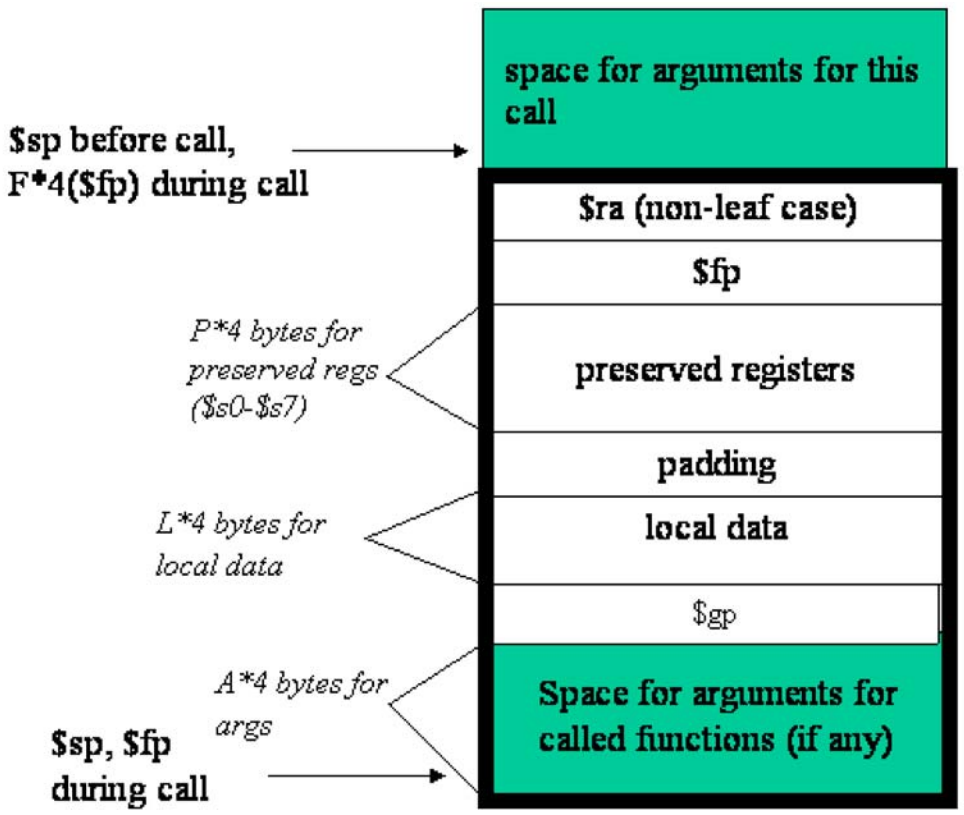
CS641 Rules for the MIPS C-compiler call convention, from http://www.cs.ucsb.edu/~franklin/30/spim/BookCallConvention.htm with a few edits to be consistent with our Gnu mips-gcc (and optimized mips-gcc-O2)

Stack allocation: The black box is around the stack frame for the current call. It contains various things saved for the current function being called, plus the green part that represents storage required for passing arguments to the functions that this function calls. Similarly, this function has access to the green part above its call frame to handle its own arguments. If there are four or fewer arguments, they are just passed in $a0-$a3, but still get space on the stack (for the convenience of the called function, you might say.) As usual, the memory addresses increase as you go up the drawing.



A is the maximum number of arguments of called functions for which this function needs to allocate space (see non-leaf function section). For non-leaf functions, A is rounded up to 4. A = 0 for leaf functions.

In fact, it's a little hard to get examples that have both L > 0 and P > 0. Optimized C "mips-gcc-O2 –S foo.c" uses preserved registers rather than stack locations, and dispenses with $fp handling. Un-optimized C "mips-gcc –S foo.c" doesn't bother with preserved registers and just uses locations on the stack accessed with $fp. The padding may occur in more than one place, as data structures on the stack are double-word aligned.

When to do things:

Rule 1: The first instruction in the function manipulates $sp to allocate F words of stack.
`addi $sp, $sp, -F*4` where F = A + L + P + (1 for $ra if needed) + (1 for $fp if in use) + (2 for $gp) + padding. F is the frame size in words, and should be even.

Corollary 1: $sp is not changed at any other point than the first and last instructions in the function.

Rule 2: If you want to use any preserved register in any place during the function, you must store them into the stack when you **enter** the function (not when you use it). Let's assume we are saving s0, s1, and s2 registers, as well as $ra and $sp. Then we would write:

sw $ra, (F-1)*4 ($sp) (not needed for leaf functions)
sw $fp, (F-2)*4 ($sp)
sw $s2, (F-3)*4($sp)
sw $s1, (F-4)*4($sp)
sw $s0, (F-5)*4 ($sp)

Rule 3. After saving preserved registers, copy $sp into $fp, so that from now until you start returning, you can use offsets from $fp to access the stack locations. Save and restore local variables as needed after this point, using $fp and the spots already allocated for them on the stack.

Rule 4: You may pass information **into** a function through **only $a0-$a3 and extra stack arguments.** As the called function, you may not use the value held in any other register except $sp, $fp, $gp and $ra (i.e. don't try to use $v0, $s0). The #4 (fifth) argument to the current function is available at (F+4)*4($fp), the #5 at (F+5)*4($fp), etc. You may store argument 0 from $a0 into its reserved spot at F*4($fp), arg1 into (F+1)*4($fp), etc., if you want.

Rule 5: You may pass information **out of** a function through **only $v0-$v1**, not $a0-$a3.

Rule 6: You may only enter a function at its beginning. You may not jal into the middle of a function.

Rule 7: You may have only one jr $ra in each function. It is the last instruction in the function, following the instructions that restore the registers saved as in Rule 2.

Extra rules for non-leaf functions:

Rule 8: Look at the declarations for all procedures this function *might* call. Find the one with the largest number of arguments, `A`, and then make A at least 4. You must allocate `A` *4 bytes upon entry into your function.

Rule 9:

Passing only up to 4 arguments: Place the arguments in $a0 - $a3

Passing more than 4 arguments: Place the first 4 arguments in $a0 - $a3. Every argument *past the 4th* is placed on the stack.

Suppose the 6 arguments are in s0-s5:

move $a0, $s0 # and 0($fp) is reserved for this on the stack
move $a1, $s1 # and 4($fp) is reserved for this on the stack
move $a2, $s2 # and 8($fp)...
move $a3, $s3 # and 12($fp)...
sw $s4, 16($fp) # store 5th argument in the stack!!! Always at this location.
sw $s5, 20($fp) # store 6th argument in the stack, always at this location.

Rule 10: You must preserve register $ra because a called-function jal destroys the value, and it is a preserved register. See Rule 2.

Example: testcall2loc.c

```
/* testing MIPS gcc calling convention */
extern int printf(char *s, ...);
int x = 6;
int main()
{
    int y = 50;
    x = mystrlen("foo", 100);
    y = mystrlen("foo", 100);
    printf("#1%d %d\n", x,y);
    printf("#1%d %d\n", x,y);
}
/* strlen with artificial second arg */
int mystrlen(char *s, int x)
{
    int i = 0;
    for (; *s; s++)
        i++;
    return i+x;
}
```

Analysis of testcall2loc.c. Note that in C, each function is separately compiled, independently of any other function.

```
main, unoptimized: frame of 40 bytes, 10 words:
Offsets from $fp:
36:  saved-ra
32:  saved-fp
28:
24:  y
20:
16:  gp
0-15: 4 args

mystrlen, unoptimized: frame of 24 bytes, 6 words:
20:
16: saved-fp
12:
8: i
(space for gp in 0-7, but not in use)
#instructions in body of loop: 10
#memory accesses in body of loop: 6

Above frame: spots for arguments to this function
24: arg 0 saved here
28: arg 1 saved here

main, optimized: frame of 40 bytes, 10 words:
36:  saved-ra
32:  saved-s2 (y)
28:  saved-s1 (addr of "foo")
24:  saved-s0 (&x)
20:
16:  gp
0-15: 4 args

mystrlen, optimized: frame of 0 bytes, args not saved to stack.
#instructions in body of loop: 4
#memory accesses in body of loop: 1
```

Unoptimized: set up stack, use it all the time (easy for debugger to understand)

```
mystrlen:
        .frame  $fp,24,$31      # vars= 8, regs= 1/0, args= 0, gp= 8 <--8+1*4+8=20, pad to 24
       ...
        addiu   $sp,$sp,-24       <-- allocate frame of 24 bytes
        sw      $fp,16($sp)       <-- save caller’s fp
        move    $fp,$sp           <-- establish our fp
        sw      $4,24($fp)        <-- store arg 0 above frame in reserved spot
        sw      $5,28($fp)        <-- store arg 1 above frame in reserved spot
        sw      $0,8($fp)         <-- init local variable i to 0
$L3:
        lw      $2,24($fp)        <-- load arg0 from stack, ptr to string
        lb      $2,0($2)          <-- load byte of string from memory
        beq     $2,$0,$L4
        lw      $2,8($fp)         <-- load i from stack location
        addiu   $2,$2,1
        sw      $2,8($fp)         <-- store incremented i back to stack
        lw      $2,24($fp)        <-- load arg0 = str ptr
        addiu   $2,$2,1
        sw      $2,24($fp)        <-- store incremented ptr back to stack
        b       $L3
$L4:
        lw      $3,8($fp)         <-- load i from stack
        lw      $2,28($fp)        <-- load arg1 from stack (x)
        addu    $2,$3,$2          <-- compute x + len = return value in v0
        move    $sp,$fp
        lw      $fp,16($sp)       <-- restore caller’s fp
        addiu   $sp,$sp,24        <-- pop off the frame
        j       $31               <-- jr $ra return to caller
        .end    mystrlen
```

Optimized: avoid stack use, minimize all memory references

```
mystrlen:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
       ...
        lb      $2,0($4)     <--load byte 0, directly using a0
        move    $3,$0        <-- init i in register
        b       $L8
        nop

$L10:
        addiu   $4,$4,1      <-- inc a0 = str ptr
        lb      $2,0($4)     <-- load next byte
        addiu   $3,$3,1      <-- inc i in register
$L8:
        bne     $2,$0,$L10   <-- check if byte is 0
        nop

        addu    $2,$3,$5     <-- add i and x, still in a1
        j       $31          <-- jr $ra return to caller
        nop
```