By Joshua Cantrell
jjc@cory.berkeley.edu

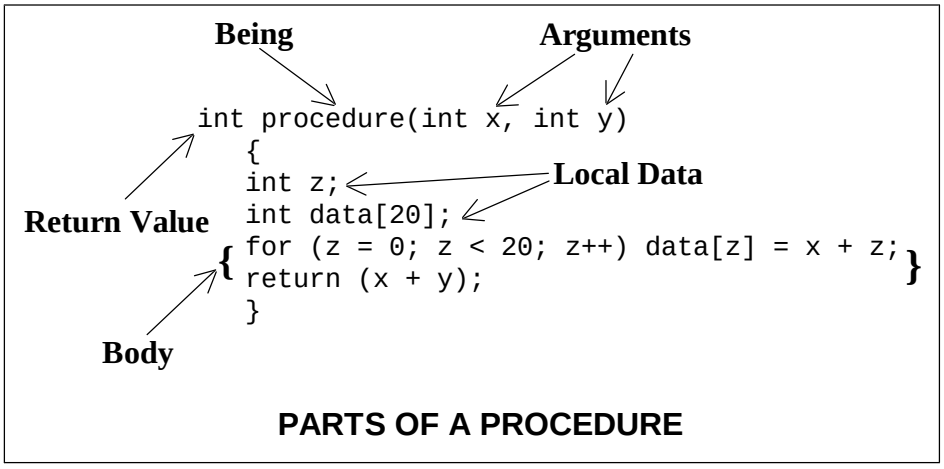
Writing MIPS Procedures

Why Write Procedures?

It never fails that there is some sequence of instructions that performs some useful job, and you'll need to use them over and over again. A procedure is a construct to reduce the number of times you must repeat these same instructions in your code. Using procedures makes your code more legible and smaller than if you used the cut and paste features of your text editor. You can even make libraries of procedures so you can reuse them in different programs.

The Basic Features of Procedures

Procedures have general features that are shared between most languages. Understanding these features will help us to understand how to represent procedures in MIPS. The main features of procedures are *arguments*, *return values*, *local data*, *body*, and *being*. The *arguments* are values passed into a procedure when it is called by some piece of code, which may or may not be in another procedure. The *return values* are values returned from the callee procedure to the caller in response to being called with particular arguments. *Local data* is data used with the procedure that exists for only as long as a particular instance of a procedure call is still being evaluated. The *being* of a procedure is the fact that it must exist somewhere in memory, so it takes up space and has a position.



Giving a Procedure a Sense of Being

In C, when you define a procedure, it's name represents a pointer to the beginning of the procedure's instructions (a procedure pointer). When programming in MIPS assembly language, it's exactly the same thing. The label that defines the beginning of the procedure's instructions *represents* the address of the first instruction.

```
procedure:                  # Address of first instruction of procedure.
           jr $ra           # Return from procedure.
```

This procedure is as simple as they come. It simply jumps back to the instruction after the one that called it using the *return address* register, $ra. When the instruction *jal* (*jump and link*), is evaluated, $ra is loaded with the address of the instruction after the *jal* instruction.

By Joshua Cantrell
jjc@cory.berkeley.edu

```
jal address             # $ra = $pc + 4;  goto address;
jr $address             # goto $address;
```

The *program counter* register, $pc, contains the current loaded instruction's location. Adding 4 to $pc gives following instruction.

Notice that *jal* takes a value representing the address. This numeric value is actually just a pointer to the first instruction of the procedure, so the label placed at the beginning of the procedure should be used there to tell the microprocessor where to jump.

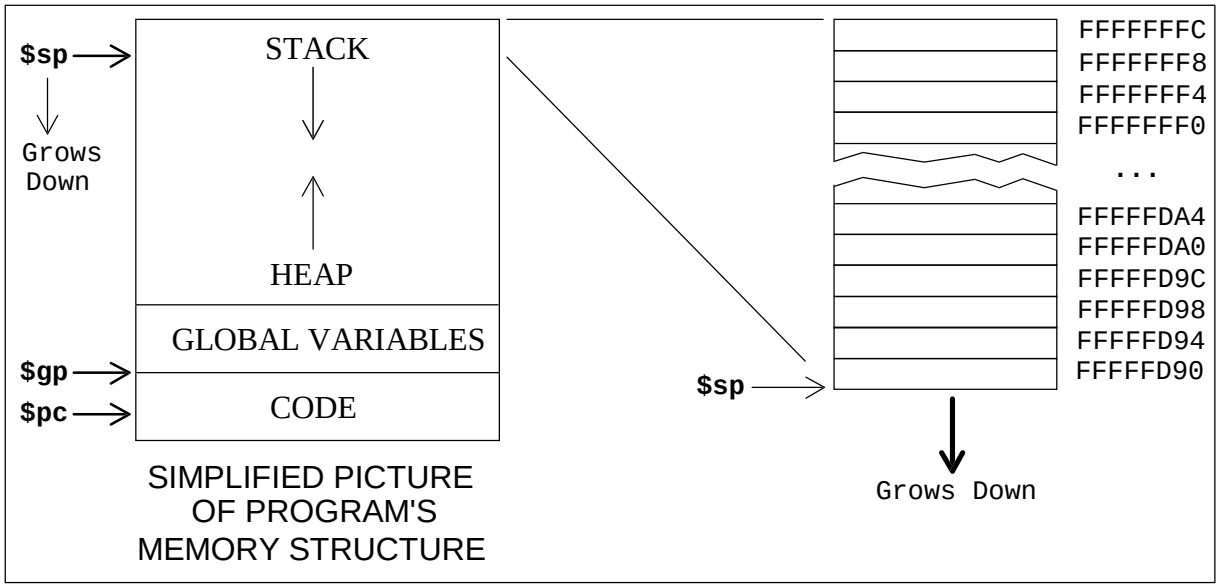
Making a Procedure Do Something

In order for a procedure to be useful, it must contain instructions. There are some conventions that these instructions must follow in order to coexist peacefully with other procedures:

1) The static registers, $s0-$s7, must have the same value leaving a procedure as coming into a procedure.
2) The address in $ra register must be available at the end of the procedure to continue proper evaluation after the procedure has ended.
3) The addresses in $sp and $fp should be the same leaving as entering.
4) The registers, $t0-$t9, $a0-$a3, and $v0-v1 may be changed without restoring their previous values at the end.

The easy part of a procedure is just the code of the body. What if we wanted to use the static registers? We would have to store the value someplace in memory to restore the value later on at the end of the procedure! This is where the *stack pointer*, $sp, comes in handy.

To make all code readable and easily understood there are a few conventions on the use of the stack pointer. The stack pointer moves down in memory as the stack grows in size, always points to the last word in the stack, and is only tampered with at the beginning and end of a procedure. It's tampered with at the beginning of a procedure to stretch the stack, and tampered with at the end of a procedure to return it to it's previous size.



By Joshua Cantrell
jjc@cory.berkeley.edu

Let's make an example of using the registers $ra, $s0, and $s1. Notice how first I allocate the space for the registers on the stack, and then I store their values onto the stack. At the end of the procedure, I restore their previous values from the stack, then deallocate the space on the stack.

```
procedure:
            addi $sp, $sp, -12      # $sp -= 3;  /*allocate 3 words*/
            sw $ra, 8($sp)          # *($sp + 2) = $ra;  /*save $ra*/
            sw $s0, 4($sp)          # *($sp + 1) = $s0;  /*save $s0*/
            sw $s1, 0($sp)          # *($sp + 0) = $s1;  /*save $s1*/
            addi $s0, $zero, 50     # $s0 = 50;
            addi $s0, $zero, -23    # $s1 = -23;
            add $a0, $s0, $s1       # $a0 = $s0 + $s1;
            jal procedure2          # procedure2($a0);
            lw $s1, 0($sp)          # $s1 = *($sp + 0); /*restore $s1*/
            lw $s0, 4($sp)          # $s0 = *($sp + 1); /*restore $s0*/
            lw $ra, 8($sp)          # $ra = *($sp + 2); /*restore $ra*/
            addi $sp, $sp, 12       # $sp += 3;  /*deallocate 3 words*/
            jr $ra                  # return;
```

The region of the procedure where you allocate room on the stack and backup register values is called the *prologue*. The region of the procedure where you restore register values and deallocate room added to the stack is called the *epilogue*. The code between these to regions is called the *body*.

```
procedure:
Prologue {  addi $sp, $sp, -12
            sw $ra, 8($sp)
            sw $s0, 4($sp)
            sw $s1, 0($sp)
Body     {  addi $s0, $zero, 50
            addi $s0, $zero, -23
            add $a0, $s0, $s1
            jal procedure2
Epilogue {  lw $s1, 0($sp)
            lw $s0, 4($sp)
            lw $ra, 8($sp)
            addi $sp, $sp, 12
            jr $ra
```

Giving a Procedure Arguments

Many procedures need extra information from the calling procedure to perform a particular job. This information is passed through the arguments. MIPS has four registers to use as arguments (4 words worth of arguments), $a0 through $a3. If more arguments are needed, then these are stored on the stack *before* the saved registers. They are found before the saved registers because the caller is the one that allocates the memory on the stack and the one that deallocates the memory after the procedure ends.

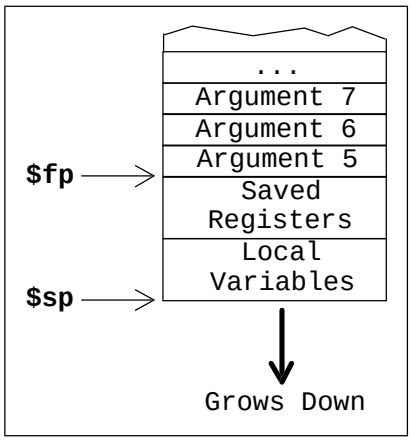
By Joshua Cantrell
jjc@cory.berkeley.edu

```
procedure:
            addi $sp, $sp, -4       # $sp -= 1;   /*allocate 1 word*/
            sw $ra, 0($sp)          # *($sp + 0) = $ra;  /*save $ra*/
            add $a0, $a0, $a1       # $a0 = $a0 + $a1;
            lw $t0, 4($sp)          # $t0 = *($sp + 1); /* Get arg on stack */
            add $a0, $a0, $t0       # $a0 = $a0 + $t0;
            jal procedure2          # procedure2($a0);
            lw $ra, 0($sp)          # $ra = *($sp + 0); /*restore $ra*/
            addi $sp, $sp, 4        # $sp += 1;   /*deallocate 1 word*/
            jr $ra                  # return;
```

Notice how an argument is obtained from the stack by looking beyond the stack space allocated in the prologue. Another way of organizing data in the stack is by using an additional pointer called the *frame pointer*, $fp. The frame pointer points to the address where the stack pointer pointed at the beginning of the procedure. Just in case the caller uses the frame pointer too, it must be saved at the beginning of the procedure. Here's an example of it's use:

```
procedure:
            addi $sp, $sp, -8       # $sp -= 2;   /*allocate 1 word*/
            sw $fp, 4($fp)          # *($sp + 1) = $fp;  /*save $fp*/
            sw $ra, 0($sp)          # *($sp + 0) = $ra;  /*save $ra*/
            addi $fp, $sp, 8        # $fp = $sp + 2;   /* set frame pointer */
            add $a0, $a0, $a1       # $a0 = $a0 + $a1;
            lw $t0, 0($fp)          # $t0 = *($fp + 0); /* Get arg on stack */
            add $a0, $a0, $t0       # $a0 = $a0 + $t0;
            jal procedure2          # procedure2($a0);
            lw $ra, 0($sp)          # $ra = *($sp + 0); /*restore $ra*/
            lw $fp, 4($fp)          # $fp = *($sp + 1); /*restore $fp*/
            addi $sp, $sp, 8        # $sp += 2;   /*deallocate 1 word*/
            jr $ra                  # return;
```

In this case, we could get the argument from a fixed position based on the frame pointer instead of adjusting the value to take into account the newly allocated stack space. The $fp can also be used to access local data stored on the stack through the use of negative offsets.



Local and Global Variables

Looking at the diagram above, it can be seen that local variables come after the saved registers. Local variables are primarily just for saving registers when you need to call a procedure (for example saving all of the temporary registers whose values cannot be destroyed) or if you run

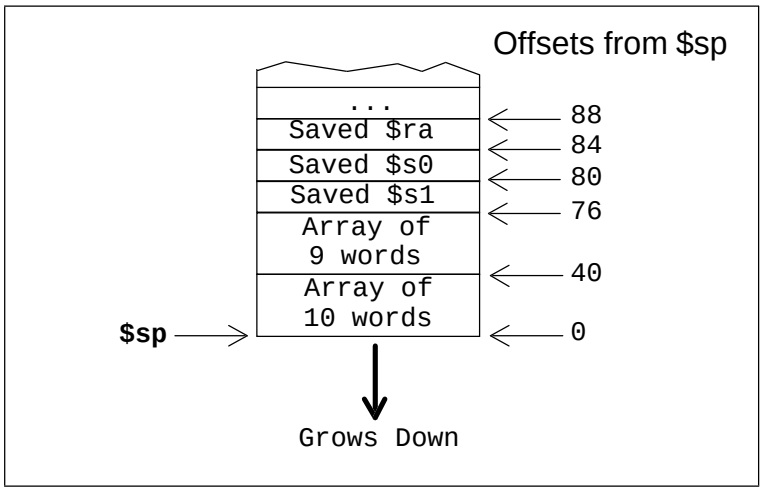
out of registers for all of the necessary variables. Allocating space on the stack for local variables is done in the *prelude*, but storing values in these variables can happen at any point in the procedure.

Global variables are located in a place in memory pointed to by the *global pointer*, $gp. All global variables are some offset from this global pointer. Variables stored in the heap are similar to global variables in that they *can* last until the end of the program, except that they aren't accessed via the global pointer, and they are created in the middle of the program (not at the very beginning) and can just as easily be destroyed in the middle of the program.

Arrays as Local Variables

Recall that when an array is declared in C, the variable is actually a pointer to the first element of that array, not the array of elements. In MIPS assembly language it's the same way. When we define a local array in a procedure, the variable that allows us to use the array is just a pointer to the very first element. For example:

```
procedure:
            addi $sp, $sp, -88      # $sp -= 22;  /* Allocate 22 words */
            sw $ra, 84($sp)         # *($sp + 21) = $ra;  /*save $ra*/
            sw $s0, 80($sp)         # *($sp + 20) = $s0;  /*save $s0*/
            sw $s1, 76($sp)         # *($sp + 19) = $s1;  /*save $s1*/
            addi $s0, $sp, 0        # $s0 = $sp + 0;  /*int $s0[10];*/
            addi $s1, $sp, 40       # $s0 = $sp + 10;  /*int $s1[9];*/
            add $a0, $zero, $s0     # $a0 = $s0;
            add $a1, $zero, $s1     # $a1 = $s1;
            jal arraywiz            # arraywiz($a0, $a1);
            lw $s1, 76($sp)         # $s1 = *($sp + 19); /*restore $s1*/
            lw $s0, 80($sp)         # $s0 = *($sp + 20); /*restore $s0*/
            lw $ra, 84($sp)         # $ra = *($sp + 21); /*restore $ra*/
            addi $sp, $sp, 88       # $sp += 22;  /*deallocate 22 words*/
            jr $ra
```



Notice that drawing a picture can help you avoid making errors when you write a procedure that must use the stack. Also be aware that the only place that the stack pointer should be decremented in is the prologue, and the only place the stack pointer should be incremented in is the epilogue. Changing the stack pointer elsewhere opens you up to errors and leads to an inefficient program.

By Joshua Cantrell
jjc@cory.berkeley.edu

A Procedure That Returns Something

Some procedures return values. MIPS reserves the $v0 and $v1 registers for these return values. This way up to a 64 bit value can be returned by a procedure. In the case that more than 64 bits must be returned to the calling procedure, it can be left on the stack. This data must be duplicated, saved, or used before another procedure is called, because that would cause the data to be overwritten. (This is typically a *bad* idea since you can easily forget not to overwrite the data, so I'm not going to show any examples.)

```
procedure:
            addi $sp, $sp, -4       # $sp -= 1;   /*allocate 1 word*/
            sw $ra, 0($sp)          # *($sp + 0) = $ra;  /*save $ra*/
            add $a0, $a0, $a1       # $a0 = $a0 + $a1;
            lw $t0, 4($sp)          # $t0 = *($sp + 1); /* Get arg on stack */
            add $a0, $a0, $t0       # $a0 = $a0 + $t0;
            jal procedure2          # $v0 = procedure2($a0);
            addi $v0, $v0, 50       # $v0 = $v0 + 50;
            lw $ra, 0($sp)          # $ra = *($sp + 0); /*restore $ra*/
            addi $sp, $sp, 4        # $sp += 1;   /*deallocate 1 word*/
            jr $ra                  # return $v0;
```