Procedure Calls in MIPS

Most high level languages include some type of procedure call functionality. Note that the terms subroutine, procedure, function and method represent similar functionality. Therefore, it is important for a machine's instruction set to offer support for procedures.

Here are the basic requirments for procedures:

Call/return mechanism.
Parameter passing mechanism. (by value and reference)
Return value mechanism.
Allocation of space for local variables.
Support for nested procedure calls.
Support for recursion.

Before looking at exactly how the MIPS architecture supports procedures it is important to consider some basic memory management concepts.

When a process begins execution, it is typically allocated memory for a number of purposes. First a section of memory is allocated to hold the code (or text), which is not modified while the process is running. A second section of memory is allocated for static data. Static variables exist during the entire execution of the process. Finally, there are two memory areas allocated which are for dynamic variables. One is called the stack and the other the heap. The stack is the area used for the temporary needs of procedures and the heap is used for other dynamic allocation needs, such as dynamic objects in C++ or Java. Often the heap and stack are actually allocated in one contiguous segment of memory. The heap will be allocated space from one end of that memory segment and the stack allocated space from the other end of that segment. Stated another way, those two structures "grow" toward each other.

MIPS Procedure Mechanisms

First of all, MIPS allocates memory exactly as described above, with an area of memory for text, static data, heap and stack.
MIPS provides the call/return mechanism through two instructions and one special register. The first instruction is "jal" which means jump and link. The operand is a label located at the beginning of the procedure. When the "jal" instruction is executed the address of the first instruction after the "jal" is stored in the "$ra" register. The "$ra" register is specifically set aside to hold the "return address". Once the return address is saved, the "jal" instruction causes a branch to the label (the beginning of the procedure). The return portion of the call/return mechanism is implemented by using the "jr" (jump register) instruction. That instruction has one operand which is a register and is nearly always the "$ra" (return address) register.

Next is parameter passing. MIPS has four registers named specifically for passing parameters (arguments). They are $a0..$a3. If you need to pass two pieces of data to a procedure, you could simply put the data into the $a0 and the $a1 registers before the procedure call and then the procedure would know how to locate the data.

Return values are handled the same as parameters except the data goes from the procedure back to the caller and the data is placed in either the $v0 or the $v1 registers.

Local variables (simple variables, not arrays) can be stored in general purpose registers such as the “$s” or “$t” registers.

Issues Related to Procedures

The mechanisms just described work beautifully in most cases. There are some limitations that an assembly language programmer must understand. Here is a brief overview of those issues.

Nested Procedures

The MIPS architecture has only one “$ra” register. If one procedure calls another and that one calls still another, what is in the “$ra” register. Clearly, it cannot hold the return addresses for multiple procedures. What must occur is that before any procedure **calls** another procedure, it must save a copy of its return address. By convention, every procedure is responsible for saving its own return address. If a procedure calls no other procedures, the return address simply remains in the “$ra” register.

If the contents of the “$ra” does need to be saved, where can that be done. The answer is on the stack. Remember, the stack is memory available for temporary storage of information. A stack has two standard operations. “Push” refers to adding something to the “top” of the stack and “pop” refers to removing something from the “top” of the stack. A stack is therefore a “Last in, First out” structure. The last thing pushed onto the stack is the first thing popped off. If a procedure is going to call another procedure, it pushes the return address onto the stack before the call and pops the return address off the stack after the call.

Notice that this approach also works for recursive procedures.

Local variables that cannot fit into registers can also be allocated space on the stack. That most often is arrays.

Should a procedure have more parameters than will fit into the $a registers or more local variables than can go into registers, the stack may again be used.

Notice that it is the responsibility of each procedure to insure that when the procedure completes, the stack is in exactly the same state as when it began. In other words, each procedure must clean up its own “mess”.

One final important point. Remember that registers are hardware devices. Each processor has only one set of registers. If “main” has a value stored in the “$t0” register and then calls a procedure, there is a possibility that the procedure may place a value into the “$t0” register, destroying the value placed there by “main”. Clearly that could be problematic. To avoid such an occurrence, MIPS uses a convention for handling general purpose registers. The convention is as follows:

The “$s” registers are considered “saved” registers. The “$t” registers are considered “temporary” registers. What that means is that (by convention) any data stored in a “saved”register will not be destroyed by a procedure. Data in “temporary” registers **may** be destroyed by a procedure. Therefore, when you are writing an assembly language procedure for the MIPS processor, if you need to use the “$s” registers, you must save their contents before use and restore their contents after use. The stack is one place you can use to store that data temporarily. Therefore, if you are writing a procedure in MIPS that does not call another procedure, it is simplest to use the temporary registers rather than the saved registers. If your procedure calls another procedure and you want to insure that your data is not destroyed by the procedure you call, you must use the “saved” registers, which means you must preserve them.