2.7 Supporting Procedures in Computer Hardware

A **procedure** or function is one tool C or Java programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time, with parameters acting as a barrier between the procedure and the rest of the program and data, allowing it to be passed values and return results. We describe the equivalent in Java at the end of this section, but Java needs everything from a computer that C needs.

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a "need to know" basis, so the spy can't make assumptions about his employer.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Place parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Place the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention in allocating its 32 registers for procedure calling:

- `$a0`–`$a3`: four argument registers in which to pass parameters
- `$v0`–`$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register `$ra`. The **jump-and-link instruction** (`jal`) is simply written

jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).

```
    jal ProcedureAddress
```

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link," stored in register `$ra`, is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *PC* in the MIPS architecture, although a more sensible name would have been *instruction address register*. The `jal` instruction saves PC + 4 in register `$ra` to link to the following instruction to set up the procedure return.

program counter (PC) The register containing the address of the instruction in the program being executed

To support such situations, computers like MIPS use a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register:

```
    jr   $ra
```

The jump register instruction jumps to the address stored in register `$ra`—which is just what we want. Thus, the calling program, or **caller**, puts the parameter values in `$a0`–`$a3` and uses `jal X` to jump to procedure `X` (sometimes named the **callee**). The callee then performs the calculations, places the results in `$v0`–`$v1`, and returns control to the caller using `jr $ra`.

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Using More Registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the Hardware Software Interface section on page 58.

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where old register values are found. The **stack pointer** is adjusted by one word for each register that is saved or restored. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a *push*, and removing data from the stack is called a *pop*.

stack A data structure for spilling registers organized as a last-in-first-out queue.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.

MIPS software allocates another register just for the stack: the stack pointer (`$sp`), used to save the registers needed by the callee. By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a C Procedure That Doesn't Call Another Procedure

EXAMPLE

Let's turn the example on page 51 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

What is the compiled MIPS assembly code?

ANSWER

The parameter variables g, h, i, and j correspond to the argument registers $a0, $a1, $a2, and $a3, and f corresponds to $s0. The compiled program starts with the label of the procedure:

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 51, which uses two temporary registers. Thus, we need to save three registers: $s0, $t0, and $t1. We "push" the old values onto the stack by creating space for three words on the stack and then store them:

```
addi $sp,$sp,-12 # adjust stack to make room for 3 items
sw   $t1, 8($sp)  # save register $t1 for use afterwards
sw   $t0, 4($sp)  # save register $t0 for use afterwards
sw   $s0, 0($sp)  # save register $s0 for use afterwards
```

Figure 2.14 shows the stack before, during, and after the procedure call. The next three statements correspond to the body of the procedure, which follows the example on page 51:

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

To return the value of f, we copy it into a return value register:

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
lw  $s0, 0($sp) # restore register $s0 for caller
lw  $t0, 4($sp) # restore register $t0 for caller
lw  $t1, 8($sp) # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr  $ra    # jump back to calling routine
```

In the example above we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- `$t0`–`$t9`: 10 temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0`–`$s7`: 8 saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller (procedure doing the calling) does not expect registers `$t0` and `$t1` to be preserved across a procedure call, we can drop two stores and two loads from the code. We still must save and restore `$s0`, since the callee must assume that the caller needs its value.

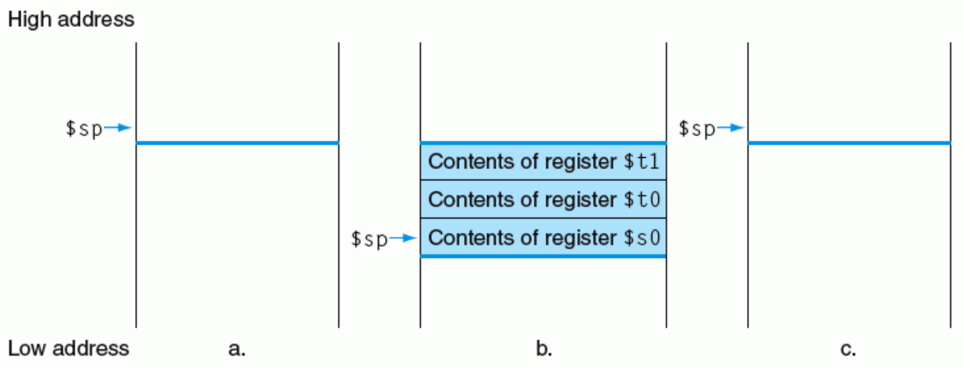

FIGURE 2.14 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Nested Procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register `$a0` and then using `jal A`. Then suppose that procedure A calls procedure B via `jal B` with an argument of 7, also placed in `$a0`. Since A hasn't finished its task yet, there is a conflict over the use of register `$a0`. Similarly, there is a conflict over the return address in register `$ra`, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved onto the stack, just as we did with the saved registers. The caller pushes any argument registers (`$a0`–`$a3`) or temporary registers (`$t0`–`$t9`) that are needed after the call. The callee pushes the return address register `$ra` and any saved registers (`$s0`–`$s7`) used by the callee. The stack pointer `$sp` is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

EXAMPLE

Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
    if (n < 1) return (1);
          else return (n * fact(n-1));
}
```

What is the MIPS assembly code?

ANSWER

The parameter variable n corresponds to the argument register $a0. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and $a0:

```
fact:
    addi  $sp,$sp,-8  # adjust stack for 2 items
    sw    $ra, 4($sp) # save the return address
    sw    $a0, 0($sp) # save the argument n
```

The first time fact is called, sw saves an address in the program that called fact. The next two instructions test if n is less than 1, going to L1 if n ≥ 1.

```
      slti  $t0,$a0,1      # test for n < 1
      beq   $t0,$zero,L1   # if n >= 1, go to L1
```

If n is less than 1, fact returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in $v0. It then pops the two saved values off the stack and jumps to the return address:

```
    addi  $v0,$zero,1 # return 1
    addi  $sp,$sp,8   # pop 2 items off stack
    jr    $ra         # return to after jal
```

Before popping two items off the stack, we could have loaded $a0 and $ra. Since $a0 and $ra don't change when n is less than 1, we skip those instructions.

If n is not less than 1, the argument n is decremented and then fact is called again with the decremented value:

```
L1: addi$a0,$a0,-1 # n >= 1: argument gets (n - 1)
    jalfact        # call fact with (n - 1)
```

The next instruction is where fact returns. Now the old return address and old argument are restored, along with the stack pointer:

```
 lw   $a0, 0($sp)  # return from jal:restore argument n
 lw   $ra, 4($sp)  # restore the return address
 addi $sp, $sp,8   # adjust stack pointer to pop 2 items
```

Next, the value register `$v0` gets the product of old argument `$a0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
mul  $v0,$a0,$v0   # return n * fact (n - 1)
```

Finally, `fact` jumps again to the return address:

```
jr   $ra           # return to the caller
```

Hardware Software Interface

A C variable is a location in storage, and its interpretation depends both on its *type* and *storage class*. Types are discussed in detail in Chapter 3, but examples include integers and characters. C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword `static`. The rest are automatic. To simplify access to static data, MIPS software reserves another register, called the **global pointer**, or `$gp`.

global pointer The register that is reserved to point to static data.

Figure 2.15 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack. The stack above `$sp` is preserved simply by making sure the callee does not write above `$sp`; `$sp` is itself preserved by the callee adding exactly the same amount that was subtracted from it, and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there. These actions also guarantee that the caller will get the same data back on a load from the stack as it put into the stack on a store because the callee promises to preserve `$sp` and because the callee also promises not to modify the caller's portion of the stack, that is, the area above the `$sp` at the time of the call.

Preserved	Not preserved
Saved registers: `$s0`–`$s7`	Temporary registers: `$t0`–`$t9`
Stack pointer register: `$sp`	Argument registers: `$a0`–`$a3`
Return address register: `$ra`	Return value registers: `$v0`–`$v1`
Stack above the stack pointer	Stack below the stack pointer

FIGURE 2.15 What is and what is not preserved across a procedure call. If the software relies on the frame pointer register or on the global pointer register, discussed in the following sections, they are also preserved.