Compiling a C Procedure That Doesn't Call Another Procedure

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?

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.

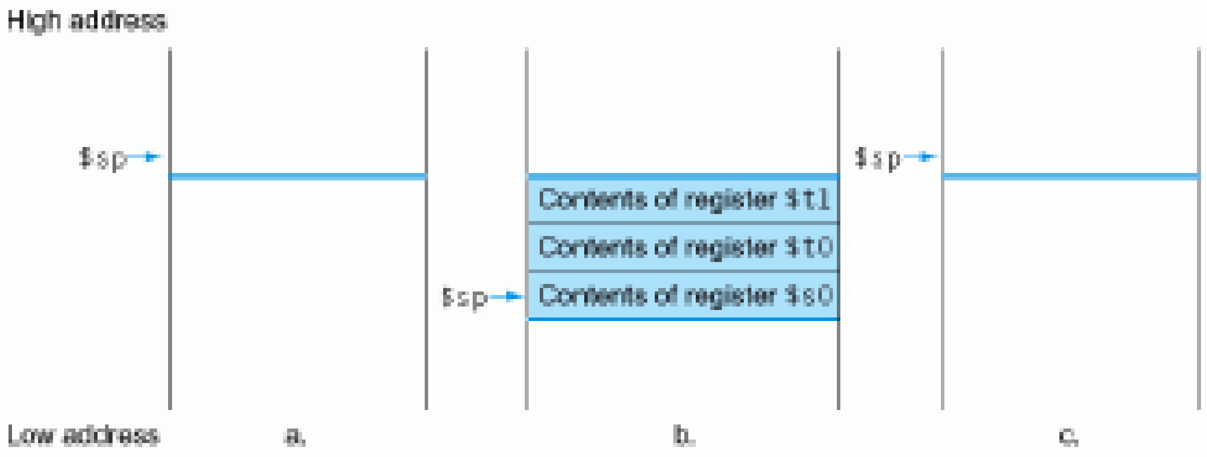

FIGURE 2.14 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.