Computing Factorial Numbers

Andreas Klappenecker

September 15, 2004

Factorial Numbers. If you have n different objects, then you can arrange them in $n \times (n-1) \times \cdots \times 2 \times 1$ ways. This number is called n factorial and is usually written as $n!$. We give a simple example of a recursive MIPS assembly language program that computes this number.

Our little program has the following structure:

1a ⟨*fac.asm* 1a⟩≡

```
⟨string definitions 2c⟩
        .text
        .globl main
⟨factorial procedure 1b⟩
⟨main procedure 2d⟩
```

In the main procedure, we prompt the user to input an integer $n \geq 0$, call the factorial procedure `fac` with the argument n, and output the result. We present the program in the literate programming style, where ⟨*chunk*⟩ represents some chunk of code that is explained in this document right after ⟨*chunk*⟩ ≡.

Calculation. If the input argument n is 0, then we return the result 1; otherwise, we recursively calculate $n!$ by the formula $(n-1)! \times n$. The procedure assumes that the input argument is contained in the register `$a0`, and the result is stored in `$v0`.

1b ⟨*factorial procedure* 1b⟩≡ (1a)

```
fac:    bne $a0, $zero, gen     # if $a0<>0, goto generic case
        ori $v0, $zero, 1       # else set result $v0 = 1
        jr  $ra                 # return
gen:    ⟨save registers 2a⟩
        addiu $a0, $a0, -1      # $a0 = n-1
        jal fac                 # $v0 = fac(n-1)
        ⟨restore registers 2b⟩
        mul $v0, $v0, $a0       # $v0 = fac(n-1) x n
        jr  $ra                 # return
```

In a recursive procedure, we need to save the register `$ra` that contains the return address before making the recursive procedure call, and restore the content of this register afterwards. In addition, we save the argument `$a0` onto the stack; therefore, after restoring the registers, we can be sure that the register `$a0` contains again the value n. The code to save the two registers is given by

2a ⟨*save registers* 2a⟩≡ (1b)

```
addiu $sp, $sp, -8      # make room for 2 registers on stack
sw  $ra, 4($sp)         # save return address register $ra
sw  $a0, 0($sp)         # save argument register $a0=n
```

and the code to restore the two registers by

2b ⟨*restore registers* 2b⟩≡ (1b)

```
lw  $a0, 0($sp)         # restore $a0=n
lw  $ra, 4($sp)         # restore $ra
addiu $sp, $sp, 8       # multipop stack
```

This example illustrates that recursive procedures are not difficult to implement in the MIPS assembly language.

Main procedure. It remains to provide some simple user interaction. The main procedure asks the user to input a nonnegative integer n; a call to the procedure `fac` performs the calculation. Finally, we print the resulting integer $n!$ and a newline.

The strings that are used in our main procedure are defined by

2c ⟨*string definitions* 2c⟩≡ (1a)

```
        .data
en:     .asciiz "n = "
eol:    .asciiz "\n"
```

Using these string definition, we can formulate the main procedure as follows:

2d ⟨*main procedure* 2d⟩≡ (1a)

```
main:   la  $a0, en             # print "n = "
        li $v0, 4               #
        syscall                 #
        li $v0, 5               # read integer
        syscall                 #
        move $a0, $v0           # $a0 = $v0
        jal fac                 # $v0 = fib(n)
        move $a0, $v0           # $a0 = fib(n)
        li $v0, 1               # print int
        syscall                 #
        la $a0, eol             # print "\n"
        li $v0, 4               #
        syscall                 #
```

That's it! It is a valuable exercise to implement an iterative algorithm to compute factorial numbers. You should try to implement several recursive functions until you feel comfortable with the register conventions and stack manipulations.