Fibonacci Numbers

An Exercise in Assembly Language Programming

Andreas Klappenecker

September 5, 2004

1 Introduction

Leonardo Pisano (1170–1250) was a scholar born in Pisa in Italy, who is probably better known by his nickname Fibonacci. During the first 30 years of his live, he traveled extensively through North Africa, where he learned numerous mathematical skills of Arab origin. In 1202, Fibonacci returned to Pisa and wrote his famous book on arithmetic and algebra, the *Liber abaci*. The book was very influential on the further mathematical development of Europe, despite the fact that it had to be copied by hand (the printing press was not introduced until the 15th century).

Among many other things, the book contained the following famous problem:

> A man puts a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year, assuming that every month each pair begets a new pair which from the second month on becomes productive?

The problem gave rise to the Fibonacci sequence $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots)$ in which each term is the sum of the previous two, so it is determined by the recurrence $f(n) = f(n-1) + f(n-2)$ with $f(0) = 0$ and $f(1) = 1$. Fibonacci could not possibly foresee the tremendous impact of his little rabbit sequence. Numerous curious applications are listed in the Neil Sloane's encyclopedia of integer sequences,

http://www.research.att.com/~njas/sequences/index.html,

and countless papers about this sequence have been published in *The Fibonacci Quarterly*.

We derive an assembly language program that calculates the Fibonacci numbers. The program is specified in Don Knuth's *Literate Programming* style that makes a program more readable for humans. Literate programming can be applied for any programming language, but it is especially nice for assembly language programs that are hard to understand without comments.

The basic philosophy of literate programming is that a file contains the documentation and the program code. The documentation produces a LaTeX or plain TeX document, and this note is an example.

In literate programming, we use a simple macro mechanism that refers to a chunk of code in the form ⟨*chunk*⟩. The particular meaning of ⟨*chunk*⟩ is specified in some other part of this document, following the statement ⟨*chunk*⟩ ≡. If a text is very long, then the chunk can be defined is smaller parts, where ⟨*chunk*⟩+ ≡ represents some code that is appended to a previous part of ⟨*chunk*⟩.

In this example, I have used Norman Ramsey's noweb tools to extract the documentation and assembly code from the file `fibonacci.nw`. A short program cannot fully illustrate the strengths of this method, but for long programs there is little doubt that a structured programming approach in the literate programming style yields excellent results. An impressive example is the documentation of Don Knuth's plain TEX that comprises a whole book. Knuth is quite confident that literate programming produces solid code, and offers $327.68 for each bug in TEX! Some well-known software companies could immediately declare bankruptcy if they would offer similar rewards.

2 The Fibonacci Program

The overall structure of our program is pretty simple, and you will figure out the details of literate programming by skimming through the remaining paragraphs. We have a single file `fib.asm` that contains all parts of the MIPS assembly language program.

An assembly language program starts with a `.text` directive which indicates that the subsequent text is some assembly code, and a declaration of a global label `main` so that the SPIM simulator finds the entry point of your program. The file `fib.asm` is structured as follows:

2a ⟨*fib.asm* 2a⟩≡

```
        .text
        .globl main
⟨output routines 3a⟩
⟨Fibonacci procedure 3d⟩
⟨main procedure 5⟩
        .data
⟨string definitions 2b⟩
```

The program consists of a couple of output routines, a recursive procedure to calculate the Fibonacci numbers, and a main procedure that prompts the user to input an argument n, calculates $f(n)$ and then prints the result.

Data Segment. Recall that `.data` is the assembly language directive which indicates that the subsequent text will be put into the data segment. We simply define here the strings

2b ⟨*string definitions* 2b⟩≡ (2a)

```
eol:    .asciiz "\n"
en:     .asciiz "n = "
fibn:   .asciiz "fib(n) = "
```

The strings are used for user interaction. The remaining three chunks of code are explained in detail in the subsequent paragraphs. You might have notice that we explained the strings first, although they are the last part of the program. You do not need to worry about the particular order, since the code extraction program `notangle` just needs to known what the code chunks mean; it will get the order right. In the extracted assembly code, the three strings will directly follow the `.data` directive, as intended.

Output routines. Let us start with some simple routines for user interaction. The first procedure prints the NUL terminated string that starts at the address contained in the register `$a0`. You might recall that the system call number 4 provided by the SPIM environment solves this task. The advantage is that `jal print_str` is more memorable than some nameless system call.

3a ⟨*output routines* 3a⟩≡ (2a) 3b▷

```
print_str:
        li $v0, 4               # print string at ($a0)
        syscall                 #
        jr $ra                  # return;
```

Recall that the string `"\n"` can be found at address `eol`. The next procedure allows you to print a line break.

3b ⟨*output routines* 3a⟩+≡ (2a) ◁3a 3c▷

```
print_eol:
        la $a0, eol             # print "\n"
        jal print_str           #
        jr $ra                  # return;
```

The `print_int` procedure prints the integer that is contained in register `$a0`.

3c ⟨*output routines* 3a⟩+≡ (2a) ◁3b

```
print_int:
        li $v0, 1               # print integer ($a0)
        syscall                 #
        jr $ra                  # return;
```

Fibonacci procedure. The calculation of the Fibonacci number $f(n)$ is done by calling the procedure `fib` with the argument n stored in the register `$a0`. Our implementation is a recursive procedure that consists of three parts:

3d ⟨*Fibonacci procedure* 3d⟩≡ (2a)

```
fib:
⟨save registers 4a⟩
⟨calculation 4b⟩
⟨restore registers 4c⟩
```

We save the registers on the stack, calculate $f(n)$ by adding $f(n-1)$ and $f(n-2)$ in the generic case, restore the registers and return the result.

The main calculation uses the registers `$s0` and `$a0`; these registers and `$ra`, the register containing the return address, need to be saved on the stack before we can proceed further.

4a ⟨*save registers* 4a⟩≡ (3d)

```
        sub $sp,$sp,12          # save registers on stack
        sw $a0, 0($sp)          # save $a0 = n
        sw $s0, 4($sp)          # save $s0
        sw $ra, 8($sp)          # save $ra to allow rec. calls
```

Recall that the stack grows from large addresses to smaller addresses. By subtracting 12 from the stack pointer we make room to save the three registers onto the stack. After the calculation, we will pop the stored values of `$a0`, `$s0`, and `$ra` from the stack and restore their values.

The argument is contained in the register `$a0`. If this register contains 0 or 1, then we can return this value, since $f(0) = 0$ and $f(1) = 1$. If `$a0` contains a value $n > 1$, then we calculate $f(n-1)$ store the value in `$s0`, then calculate $f(n-2)$, add the two values and return the result.

4b ⟨*calculation* 4b⟩≡ (3d)

```
        bgt $a0,1, gen          # if n>1 then goto generic case
        move $v0,$a0            # output = input if n=0 or n=1
        j rreg                  # goto restore registers

gen:    sub $a0,$a0,1           # param = n-1
        jal fib                 # compute fib(n-1)
        move $s0,$v0            # save fib(n-1)

        sub $a0,$a0,1           # set param to n-2
        jal fib                 # and make recursive call
        add $v0, $v0, $s0       # $v0 = fib(n-2)+fib(n-1)
```

It remains to restore the values of the registers that we have changed. For example, we have reduced the value of `$a0` twice during the calculation, so we need to restore it now. Similarly, we have modified `$s0`; by convention, we need to restore it to its original value.

4c ⟨*restore registers* 4c⟩≡ (3d)

```
rreg:   lw  $a0, 0($sp)         # restore registers from stack
        lw  $s0, 4($sp)         #
        lw  $ra, 8($sp)         #
        add $sp, $sp, 12        # decrease the stack size
        jr $ra
```

Main procedure. The main procedure is fairly straightforward. We prompt the user to input the argument n, call the Fibonacci procedure `fib` with this argument, and print the result.

5 ⟨*main procedure* 5⟩≡ (2a)

```
main:
        la $a0, en      # print "n = "
        jal print_str

        li $v0, 5       # read integer
        syscall         #

        move $a0, $v0   # $a0 := $v0
        jal fib         # call fib(n)
        move $s0, $v0   # store result in $s0

        la $a0, fibn
        jal print_str

        move $a0,$s0    # print result
        jal print_int   #
        jal print_eol   #

        li $v0,10       # exit
        syscall         #
```

The system call 10 is the proper way to exit from SPIM.

Final Remark. Many people tend to get religious about how one should develop and document software. I do not try to convince you that this is "the" way to do it. Personally, I enjoy reading literate programs and find the typesetting of LaTeX more pleasant than other alternatives. In my experience, the documentation and code needs to be in the same file, otherwise the two documents then the get out of sync. You should try the literate programming style yourself. The site www.literateprogramming.com contains numerous examples, tools, and manuals.