CS 241 - Week 2 Tutorial

Assembly Language Programming

Fall 2014

Summary

- Introduction
- How to write a MIPS loop
- How to print to standard output and Using the stack
- How to create and use procedures

Introduction

$31 is a "special" register that contains the program's *return address*. There is an instruction called `jalr` (Jump And Link Register) which allows us to modify $31 in a predictable way. From the MIPS Reference Sheet, `jalr $s` will do

```
temp = $s
$31 = pc
pc = temp
```

That is, $31 is overwritten with the current value of the pc, then we jump to the address in $s. This allows us to not only jump to a new location in the program, but also to come back when we are done. Essentially, `jalr` enables us to write procedures.

You may have noticed when playing with the MIPS emulator that $30 has an unusual value.

```
Running MIPS program.
MIPS program completed normally.
...
$29 = 0x00000000    $30 = 0x01000000    $31 = 0x8123456c
```

which, it turns out, is the amount of available RAM in the MIPS emulator. When we need extra storage beyond the 32 machine registers, we can make use of RAM. We can treat the value in $30 like the pointer to the top of a stack (a *stack pointer*). Pseudocode for stack ADT operations could then look something like this:

```
push($s):
  sw $s, -4($30)     ; store $s at the top of the stack
  lis $s
  .word 4
  sub $30, $30, $s   ; move the top of the stack

pop($s):
  lis $s
  .word 4
  add $30, $30, $s   ; move the top of the stack
  lw $s, -4($30)     ; read $s from the top of the stack
```

Note that once $s is stored onto the stack, we can safely overwrite its value immediately for use in updating the stack pointer. If 4 is sitting in some other register, then we do not need to load 4 into $s. Also, if we need to push or pop multiple values in succession, we can extend this idea so that we only need to move the stack pointer once. If after every block of pushes/pops $30 is pointing to the top of our stack, then we have successfully implemented the "program stack".

The Fibonacci numbers are a sequence of integers in which each number in the sequence is derived from the recursive formula

$$f_n = f_{n-1} + f_{n-2}$$

Where $f_0 = 0$ and $f_1 = 1$ and $n \geq 2$

Problem 1 - How to write a MIPS loop

- $1 contains a non-negative number n
- Find the n^{th} Fibonacci number and place it in $3

Problem 2 - How to create and use procedures

- Convert Problem 1 into a procedure named `fib` which expects $1 to be n and outputs the result in $3
- Apart from $3, upon return every register should contain the same value as when the procedure was called

Problem 3 - Printing to stdout and using the stack

- $1 contains an integer $n \geq 1$

- Using the procedure `fib` from problem 2, print the first n Fibonacci numbers forward, then in reverse. You may assume you are given a procedure named `print` which will print the value in $1 interpreted as a signed decimal integer. You may only compute each Fibonacci number once.

Problem 4 - Various skills

- Using the procedure `fib` from Problem 2, check if the array with starting address in $1 and number of items in $2 is the Fibonacci sequence. Put 1 in $3 if the array is the Fibonacci sequence, 0 otherwise.

1.
```
*Loop: BEQ $0, $0, loop*
```

```
fib:  lis $2
      .word 1
      add $3, $0, $0
      add $4, $2, $0
loop: beq $1 $0, end
      sub $1, $1, $2
      add $5, $4, $0
      add $4, $3, $4
      add $3, $5, $0
      beq $0, $0, loop
end:  jr $31
```

2.
```
sw $1, -4($30)
sw $2, -8($30)
sw $4, -12($30)
sw $5, -16($30)
lis $2
.word 16
sub $30, $30, $2
```

```
CallingFib:
lis $10
.word fib
push $31
jarl $10
pop $31
```

```
lis $2
.word 16
add $30, $30, $2
lw $1, -4($30)
lw $2, -8($30)
lw $4, -12($30)
lw $5, -16($30)
jr $31
```