MIPS function continued

Review

- Functions
  - Series of related instructions one after another in memory
  - Called through the jal instruction
  - Pointed to by a label like any other
  - Returns by calling jr $ra
- Stack
  - Top pointed to by $sp
  - Used to store registers between function calls
    - Typically $ra and $s0-7; but will need to store $a0-3, $v0-1, and $t0-9 in the event of a series of nested calls
  - Can be used to store other things too
    - Such as characters in a string when reversing the string

Review

- Characters
  - Use one byte of memory
  - Can be used like integers and stored within integers
  - Can use constants through the use of single quotes, e.g. li $t0, ‘a’
  - Passed into and out of memory through sb / lb
- Strings
  - An array of characters in memory
  - Used like other arrays except
    - Use .asciiz for a null-character terminated (C styled) string
    - Use .ascii for a non null-character terminated string (not recommended)
    - Enclose the array in double quotes

In Class Exercise 3

```
    .data
A:  .word 0,1,2,3,4,5,6,7,8,9
NF: .asciiz "not found"
FL: .asciiz "found in lower"
FU: .asciiz "found in upper"

    .text
    .globl main
main:
    la $a0, A
    li $a1, 10
    li $a2, 6
    li $a3, 5
    jal findElements

    la $a0, NF
    blt $v0, $0, print
    la $a0, FL
    beq $v0, $0, print
    la $a0, FU

print:
    li $v0, 4
    syscall

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

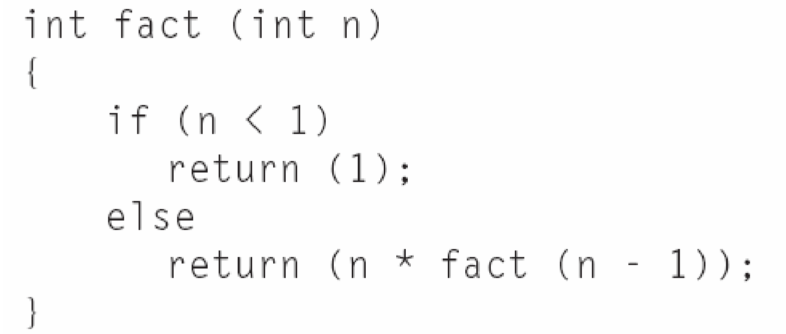
```
findElements:
    # $a0 = array
    # $a1 = length
    # $a2 = value 1
    # $a3 = value 2
    # $v0 = found value status
    addi $sp, $sp, -4   # Push ra onto stack
    sw $ra, 0($sp)

    li $v0, -1          # Set return to default
    li $v1, 0           # i
loop:
    bge $v1, $a1, find_done
    lw $t0, 0($a0)      # A[i]
    beq $t0, $a2, first_value
    beq $t0, $a3, second_value
    addi $v1, $v1, 1    # update i
    addi $a0, $a0, 4    # update array slot
    j loop
first_value:
    ori $v0, $0, 0      # Set return to 0
    j find_done
second_value:
    ori $v0, $0, 1      # Set return to 1

find_done :
    lw $ra, 0($sp)      # Pop ra off of stack
    addi $sp, $sp, 4
    jr $ra              # Return
```

Implementing a Recursive Function

- Suppose we want to implement this in MIPS:



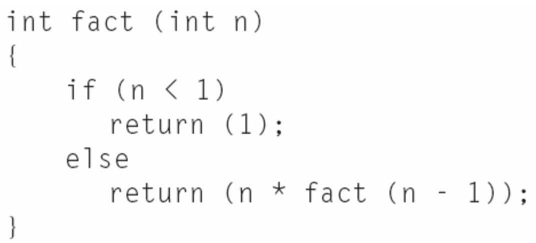
```
int fact (int n)
{
    if (n < 1)
       return (1);
    else
       return (n * fact (n - 1));
}
```

- It is a **recursive** function – a function that calls itself.
- It will keep on calling itself, with different parameters, until a terminating condition is met.

The Recursive Function

What happens if we call fact(4)?

- First time call fact, compare 4 with 1, not less than 1, **call** fact again – fact(3).
- Second time call fact, compare 3 with 1, not less than 1, **call** fact agai – fact(2).
- Third time call fact, compare 2 with 1, not less than 1, **call** fact again – fact(1).
- Fourth time call fact, compare 1 with 1, not less than 1, **call** fact again – fact(0).
- Fifth time call fact, compare 0 with 1, less than 1, **return 1.**
- Return to the time when fact(0) was called (during the call of fact(1)). Multiply 1 with 1, **return 1.**
- Return to the time when fact(1) was called (during the call of fact(2)). Multiply 2 with 1, **return 2.**
- Return to the time when fact(2) was called (during the call of fact(3)). Multiply 3 with 2, **return 6.**
- Return to the time when fact(3) was called (during the call of fact(4)). Multiply 4 with 6, **return 24**.



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

The Recursive Function

- In MIPS, we say calling a function as going to the function. So we go to the function over and over again, until the terminating condition is met.
- Here, the function is called “fact,” so we will have a line of code inside the fact function:

```
jal fact
```

```
fact:   jal fact
```

The Recursive Function

- The parameter should be passed in `$a0`. In the C function, every time we call fact, we call with n-1. So, in the MIPS function, before we do "`jal fact`", we should have "`addi $a0, $a0,-1.`"

```
fact:   addi $a0, $a0, -1
        jal fact
```

The Recursive Function

- After calling fact, we multiply the return result with n, so, need to add multiplications.

```
fact:   addi $a0, $a0, -1
        jal fact
        mul $v0, $v0, $a0
```

The Recursive Function

- After multiplying, we return.

```
fact:   addi $a0, $a0, -1
        jal fact
        mul $v0, $v0, $a0
        jr $ra
```

The Recursive Function

- So, one if else branch is done. The other branch is to compare `$a0` with 1, and should call `fact` again if less than 1 and otherwise return 1.

```
fact:   slti $t0, $a0, 1
        beq $t0, $zero, L1
        ori $v0, $0, 1
        jr $ra
L1:     addi $a0, $a0, -1
        jal fact
        mul $v0, $v0, $a0
        jr $ra
```

Any problems?

The Recursive Function

- The problem is that the function will call itself, as we have expected, but it will not return correctly!
- We need to save `$ra`, because we made another function call inside the function. We should always do so.
- Is this enough?

```
fact:   addi $sp, $sp, -4
        sw $ra, 0($sp)
        slti $t0, $a0, 1
        beq $t0, $zero, L1
        ori $v0, $0, 1
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
L1:     addi $a0, $a0, -1
        jal fact
        mul $v0, $v0, $a0
        lw $ra, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

The Recursive Function

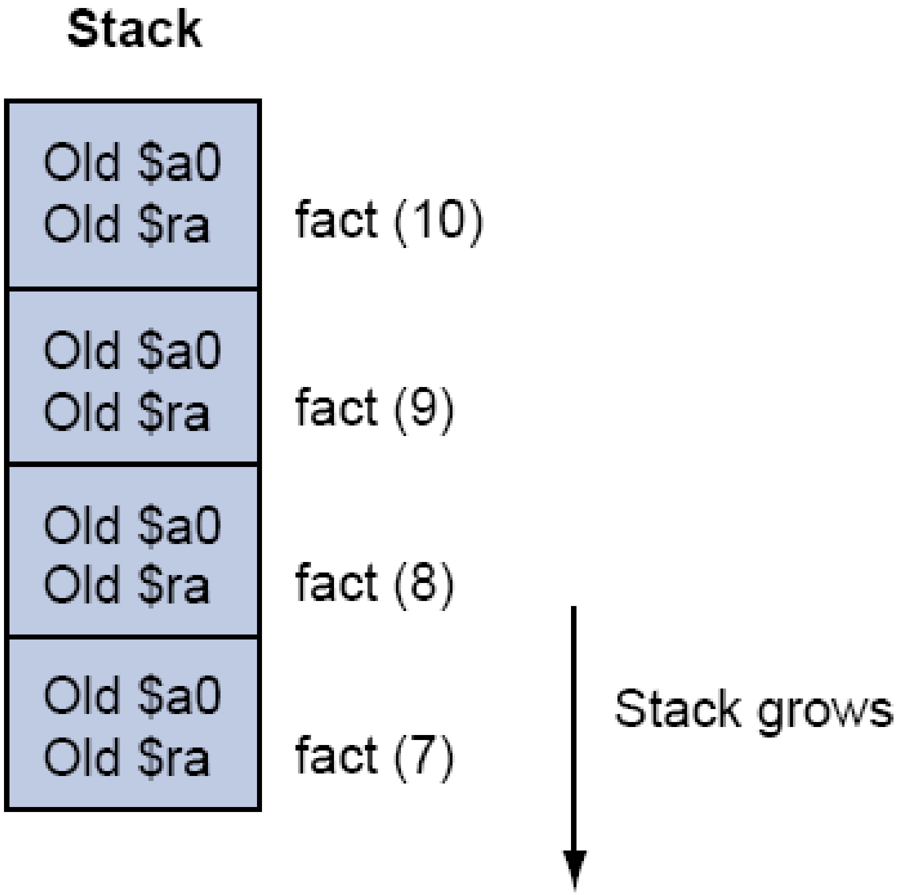
- So now we can return to the main function, but the return result is 0, why?
- A call to `fact` modifies `$a0`. But when we return from a call, we multiply it with `$a0`!
- So, should also save `$a0`!
- Restore it before using it again.

```
fact:   addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $a0, 0($sp)
        slti $t0, $a0, 1
        beq $t0, $zero, L1
        ori $v0, $0, 1
        addi $sp, $sp, 8
        jr $ra
L1:     addi $a0, $a0, -1
        jal fact
        lw $ra, 4($sp)
        lw $a0, 0($sp)
        mul $v0, $v0, $a0
        addi $sp, $sp, 8
        jr $ra
        .text
        .globl main
Main:   li $a0, 4
        jal fact

done:   li $v0,10
        syscall

fact:   addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $a0, 0($sp)
        slti $t0, $a0, 1
        beq $t0, $zero, L1
        ori $v0, $0, 1
        addi $sp, $sp, 8
        jr $ra
L1:     addi $a0, $a0, -1
        jal fact
        lw $ra, 4($sp)
        lw $a0, 0($sp)
        mul $v0, $v0, $a0
        addi $sp, $sp, 8
        jr $ra
```

The Stack During Recursion



Stack

Old $a0
Old $ra

fact (10)

Old $a0
Old $ra

fact (9)

Old $a0
Old $ra

fact (8)

Old $a0
Old $ra

fact (7)

Stack grows

Two other MIPS pointers

- `$fp:` When you call a C function, the function may declare an array of size 100 like int A[100]. It is on the stack. You would want to access it, but the stack pointer may keep changing, so you need a fixed reference. `$fp` is the "frame pointer," which should always point to the first word that is used by this function.
- `$gp:` the "global pointer." A reference to access the static data.