Announcements

- Program part A is due on Tuesday.
  - The program must be your own work.
  - If we find you have copied your solution from another student, we will bring the matter before the student honor council **(The penalties could include receiving an XF in the class or suspension or expulsion from the University)**
- Reading for this lecture:
  - finish the SPIM manual
  - Mano section 8-3 and 8-7
- Reminder that I have office hours on Wed 10-11
- Thursday is Thanksgiving (no class)

Stacks

- A LIFO data structure
  - classic analogy is a spring loaded plate holder found in dorm kitchens (top plate is always the first one removed)
- Expressions can be represented in three forms:
  - normally think of expressions as infix
    - A + B
  - prefix notation
    - + A B or + - A B C
    - the language Lisp uses this form
  - post fix
    - A B +
    - RPN calculators, Postscript language used in printers

Evaluating Expressions in RPN

- Expressions don't require parentheses
- Read input into tokens
  - an function (operator), or a number (operand)
- To evaluate an RPN expression:
  - if the current token is an operand:
    - push(operand)
  - if the current token is a unary operator:
    - pop(operand), perform operation, push(result)
  - if the current token is a binary operator:
    - pop(operand1), pop(operand2), perform operation, push(result).

Stacks are also useful for procedure calls

- call and return are LIFO operations
- use the stack to store the state of procedures
  - return address
  - local variables
    - since we need to have one copy per invocation not per function
  - temporary data
    - this is really a special case of local variables.
- Items are placed on the stack in *frames*
  - frames are multi-word groups that hold the state of one procedure invocation.

SPIM (MIPS) Stack Segment

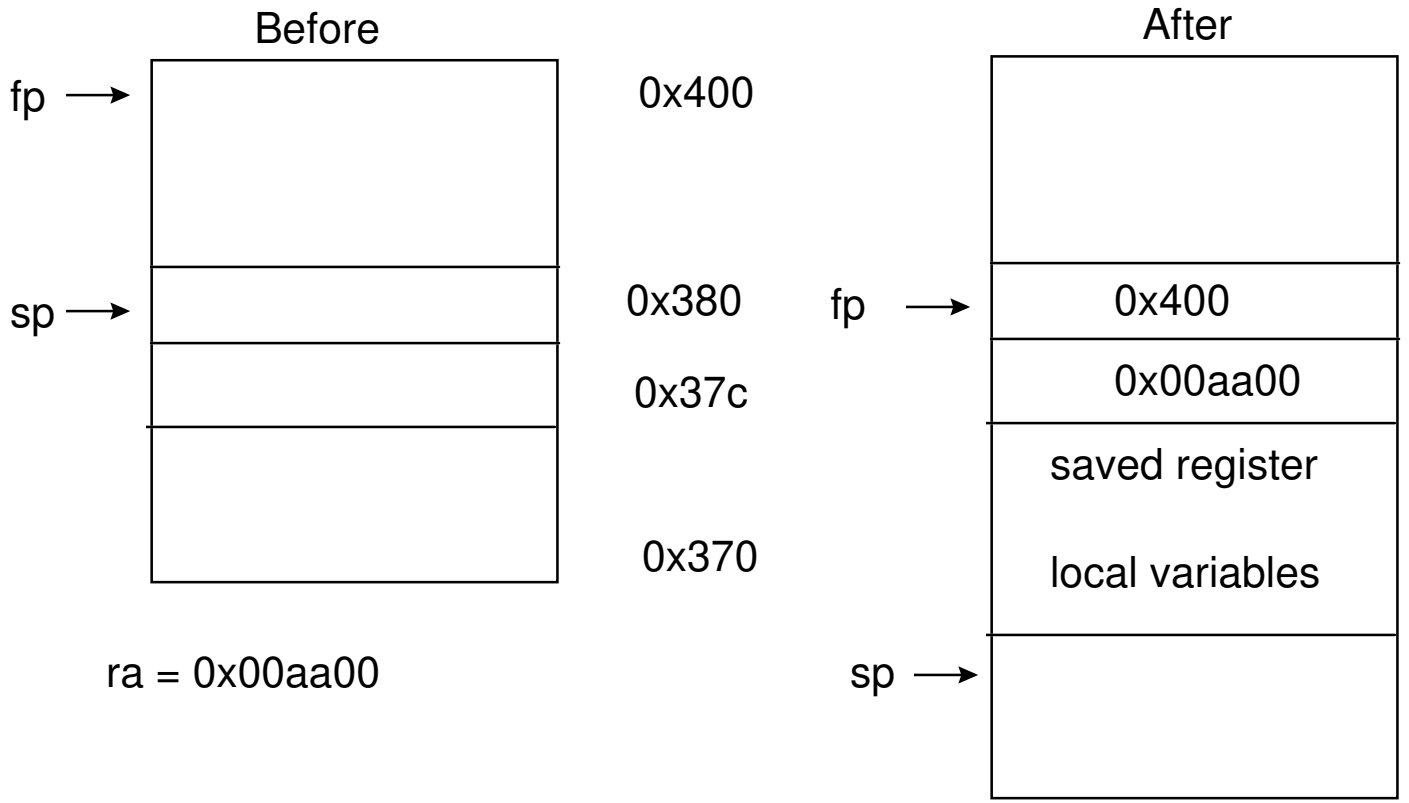
- Calling Convention
  - describes how procedures communicate information
  - caller:
    - passes first 4 arguments in registers a0-a3
    - saves registers t0-t9 if they are active (in use)
    - executes call Instruction (jal)
  - callee:
    - updates the stack pointer
    - saves registers s0-s7 if used
    - allocates the stack frame

Stack Frame

- To setup stack frame:

```
sw $fp, 0($sp)
sw $ra, -4($sp)
move $fp, $sp
subu $sp, $sp, 8
```

at least 8, but could be more

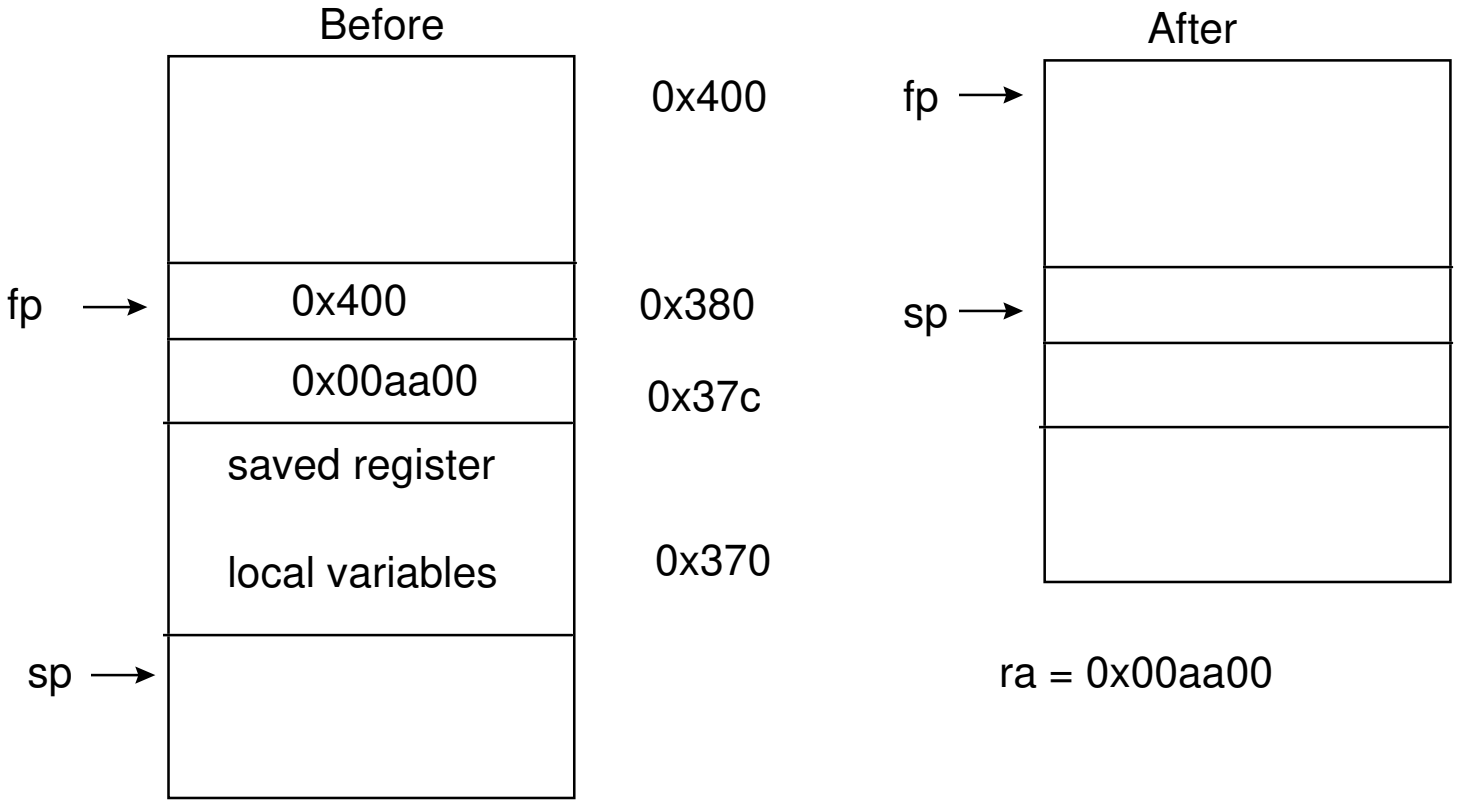


Return from Subroutine

- Cleanup stack frame:

```
lw $ra, -4($fp)
lw $fp, 0($fp)
addu $sp, $sp, 8
jr $ra
```

at least 8, but could be more



Consider Writing a Recursive Function

- Function is fib (Fibonacci number)
  - fib(1) = 1
  - fib(2) = 1
  - fib(n) = fib(n-1) + fib(n-2) for n > 2
- Hints:
  - use $a0 to pass the parameter
  - use $v0 to return result
  - Need to store temporary values on the stack:
    - need to store n (or n-2) during first call
    - need to store result of fib(n-1) during n-2 call.
- Everyone take out a sheet of paper and write fib.
  - You have about 7 min.

Fib Main routine

```
main:
        sw $fp, 0($sp)
        sw $ra, -4($sp)
        move $fp, $sp
        subu $sp, $sp, 8

mainLoop:
        la $a0, prompt
        li $v0, 4
        syscall

        li $v0, 5               # read int
        syscall

        move $a0, $v0
        jal fib
        move $s0, $v0

        la $a0, resultStr       # print resp string
        li $v0, 4
        syscall

        move $a0, $s0
        li $v0, 1
        syscall
        b mainLoop

        .data
prompt:
        .asciiz "\nEnter an integer: "
        .align 4
resultStr:
        .asciiz "Fib # = "
```

Fib Solution

```
fib:
        sw $fp, 0($sp)
        sw $ra, -4($sp)
        move $fp, $sp
        subu $sp, $sp, 16

        li $t0, 1
        ble $a0, $t0, baseFib
        subu $a0, $a0, $t0
        sw $a0, -12($fp)
        jal  fib                        # fib n-1
        sw $v0, -8($fp)

        lw $a0, -12($fp)
        li $t0, 1
        subu $a0, $a0, $t0    # fib n-2
        jal  fib

lw $t0, -8($fp)
        addu $v0, $v0, $t0
        b endFib

baseFib:
        li $v0, 1
endFib:
        move $sp, $fp
        lw $ra, -4($sp)
        lw $fp, 0($sp)
        jr $ra
```