Lecture 10: Assembly Programming Part 3

Assembly Test

- July 31, in class
  - 1 hour
  - 5% of final mark (term test was 20%)
  - Just focusing on assembly
  - Pen & paper (like exam)

Last Week

- Memory access:
  - Arrays
  - Structs
  - Alignment
  - Segments: .data , .text
- Functions:
  - Parameters
  - Stack
  - Return address
  - Calling conventions

```
.data
v1:     .word 52
a1:     .space 100

.text
la $t0, a1
lw $t2, 16($t0)
```

```
addi $sp, $sp, -4
sw $t2, 0($sp)

jal     SOME_FUNCTION

lw $t5, 0($sp)
addi $sp, $sp, 4
```

Warmup

- This functions has two parameters and two returned values

```
def sum_prod(a, b):
    s = a + b
    p = a * b
    return (s, p)
```

Calling sum_prod

- Given variables A, B, get sum_prod(A,B)
- Steps:
  - Declare variables
  - Load into registers
  - Push onto stack
  - Call function
  - Pop results from stack into registers

Calling sum_prod: declare vars

```
.data
A:      .word  7
B:      .word  5
```

Calling sum_prod: load values

```
.data
A:      .word  7
B:      .word  5

.text
main:  la $t0, A            # $t0 = address of A
       lw $t1, 0($t0)       # $t1 = value of A
       la $t2, B            # $t2 = address of B
       lw $t3, 0($t2)       # $t2 = value of B
```

Calling sum_prod: push and call

```
.data
A:      .word  7
B:      .word  5

.text
main:  la $t0, A              # $t0 = address of A
       lw $t1, 0($t0)         # $t1 = value of A
       la $t2, B              # $t2 = address of B
       lw $t3, 0($t2)         # $t2 = value of B

       addi $sp, $sp, -4      # push A onto the stack
       sw $t1, 0($sp)
       addi $sp, $sp, -4      # push B onto the stack
       sw $t3, 0($sp)
       jal sumprod            # "call" the sign function
```

Calling sum_prod: pop results

```
.data
A:      .word  7
B:      .word  5

.text
main:  la $t0, A              # $t0 = address of A
       lw $t1, 0($t0)         # $t1 = value of A
       la $t2, B              # $t2 = address of B
       lw $t3, 0($t2)         # $t3 = value of B

       addi $sp, $sp, -4      # push A onto the stack
       sw $t1, 0($sp)
       addi $sp, $sp, -4      # push B onto the stack
       sw $t3, 0($sp)
       jal sum_prod           # "call" the sign function
       lw $t5, 0($sp)         # pop the sum A+B off the stack
       addi $sp, $sp, 4
       lw $t6, 0($sp)         # pop the product A*B off stack
       addi $sp, $sp, 4
```

The Stack is FILO / LIFO

- First in, last out.
- Equivalently:
  Last in, first out (LIFO).
- If you push A, B, C (in this order)…
- …when you pop you get C, B, A

sum_prod: implementation

- To implement sum_prod
  - (decide on registers)
  - Pop arguments off stack
  - Do the computation
  - Push return values
  - Return to caller

Implement sum_prod: arguments

```
# sum_prod: (A, B) -> (A+B, A*B)
# $t0=A, $t1=B, $t3=A+B, $t4=A*B

sum_prod:     lw $t1, 0($sp)        # pop B off the top of
              addi $sp, $sp, 4      # the stack first
              lw $t0, 0($sp)        # now we pop A
              addi $sp, $sp, 4
```

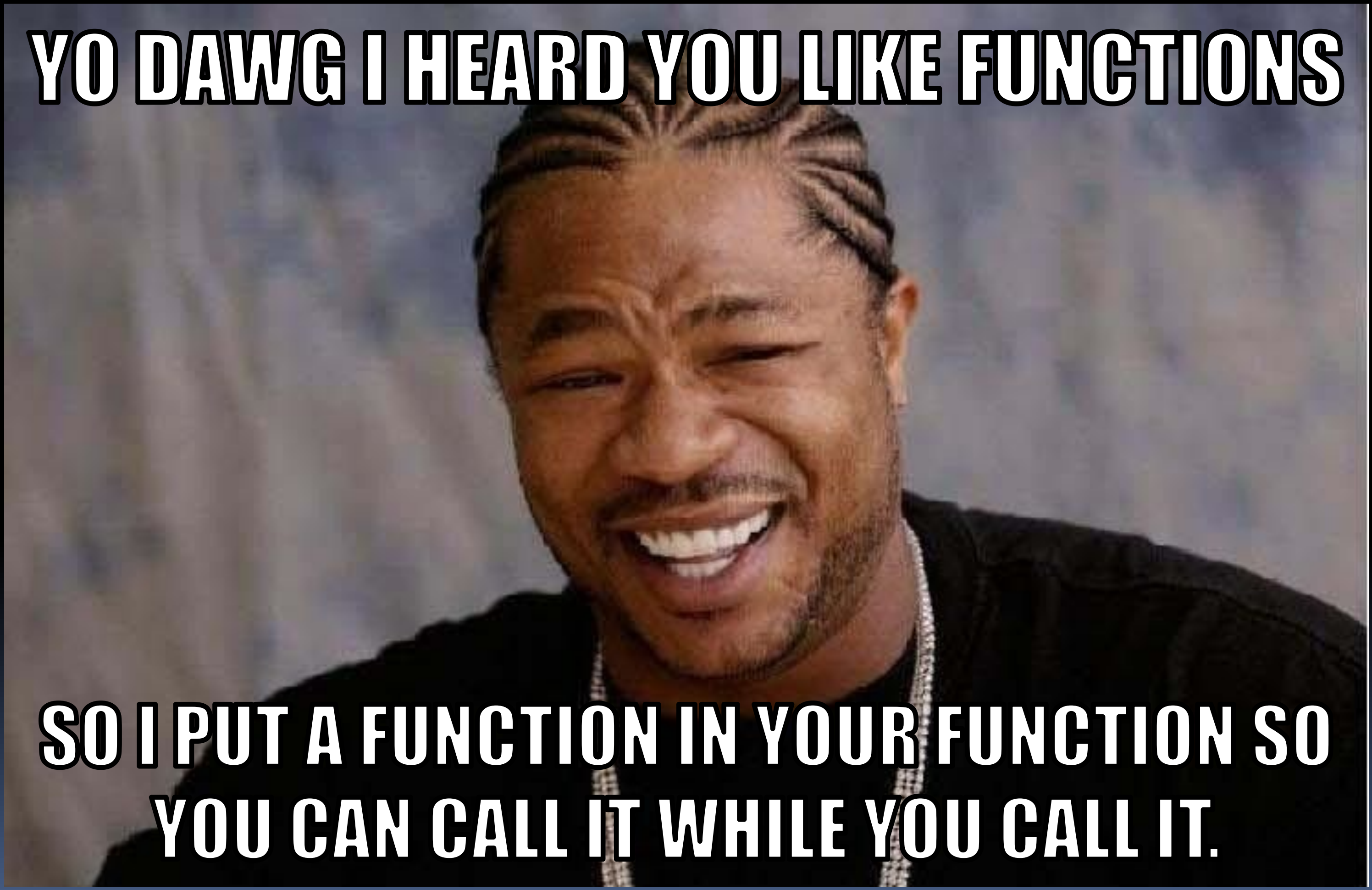
Implement sum_prod: compute

```
# sum_prod: (A, B) -> (A+B, A*B)
# $t0=A, $t1=B, $t3=A+B, $t4=A*B

sum_prod:     lw $t1, 0($sp)         # pop B off the top of
              addi $sp, $sp, 4       # the stack first
              lw $t0, 0($sp)         # now we pop A
              addi $sp, $sp, 4

              add $t3, $t0, $t1      # $t3 = A+B
              mult $t0, $t1          # compute A*B
              mflo $t4               # store the result in $t4
#(note we are assuming 32 bit result here!)
```

Implement sum_prod: return

```
# sum_prod: (A, B) -> (A+B, A*B)
# $t0=A, $t1=B, $t3=A+B, $t4=A*B

sum_prod:     lw $t1, 0($sp)         # pop B off the top of
              addi $sp, $sp, 4       # the stack first
              lw $t0, 0($sp)         # now we pop A
              addi $sp, $sp, 4

              add $t3, $t0, $t1      # $t3 = A+B
              mult $t0, $t1          # compute A*B
              mflo $t4               # store the result in $t4
#(note we are assuming 32 bit result here!)

end:          addi $sp, $sp, -4      # first push A*B on stack
              sw $t4, 0($sp)         # so it comes out second
              addi $sp, $sp, -4      # now push A+B onto stack
              sw $t3, 0($sp)         # so it comes out first

              jr $ra                 # jump back to caller
```

Functions Calling Functions



Calling From Inside Function

- Assume we already have max(a,b)
- We want to implement `max3(a,b,c)`
- Easy, just call `max` twice:
  - `tmp = max(a, b)`
  - `res = max(tmp, c)`
  - `return res`
- max pseudo code:
  - pop a, b into $t0, $t1
  - If $t0 > $t1 set $t2 = $t0 else $t2 = $t1
  - Push $t2 onto stack

max(a,b)

```
max:  lw $t1, 0($sp)              # first pop b from stack
      addi $sp, $sp, 4
      lw $t0, 0($sp)              # now pop a from stack
      addi $sp, $sp, 4

# input values are in $t0, $t1, output will be in $t2
      ble $t0,$t1, else           # if a<=b we jump to else
      add $t2, $t0, $zero         # a>b so set $t2 to $t0
      j end
else: add $t2,$t1,$zero           # a<=b so set $t2 to $t1
end:  addi $sp, $sp, -4           # push result onto stack
      sw $t2, 0($sp)
      jr $ra                      # jump back to caller
```

max3(a,b,c) in "Assembly"

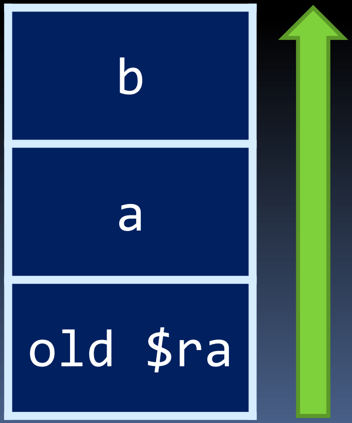
- Pop a, b, c into registers $t0, $t1, $t2
- Push $t0, $t1 onto stack
- Call max (`jal max`)
- Pop partial max into $t3
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4
- Push $t4 final max
- Return to caller (jr $ra)

Problem 1: max uses `$t2` internally

Problem 2: $ra was overwritten by `jal max`

Saving $ra

- When calling function f from inside function g we execute jal function f
- This overwrites return address $ra
- We need to preserve it, but where?
- Stack to the rescue:
  - Push old value of $ra onto the stack.
  - Push arguments for f
  - Call f
  - Pop return value
  - Pop old value of $ra



max3(a,b,c) in "Assembly"

- Pop a, b, c into registers $t0, $t1, $t2
- Push $ra
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into $t3
- Pop $ra
- Push $ra
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4
- Pop $ra
- Push $t4 final max
- Return to caller (jr $ra)

This is a little silly.

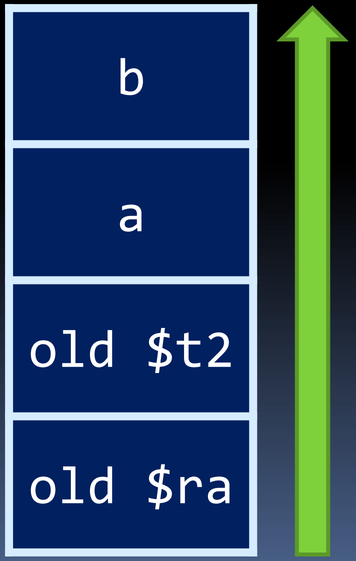
max3(a,b,c) in "Assembly"

- Pop a, b, c into registers $t0, $t1, $t2
- Push $ra
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into $t3
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4
- Pop $ra
- Push $t4 final max
- Return to caller (jr $ra)

If we need to call another function: push $ra to the stack at the beginning of the function, and pop at the end

Wait a Minute…

- We also need to preserve $t2 since it holds c
- Push it on stack along with $ra and pop it after the function returns



max3(a,b,c) in "Assembly"

- Pop a, b, c into registers $t0, $t1, $t2
- Push $ra
- Push $t2 (we need to pop $t2 before $ra!)
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into $t3
- Pop $t2
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4
- Pop $ra
- Push $t4 final max
- Return to caller (jr $ra)

Preserving Register Values

- We've already demonstrated why we'd need to push `$ra` and `$t2` onto the stack when calling function from another function.
- What about the other registers?
- How do we know that a function we called didn't overwrite registers that we were using?
  - Remember there is only one register file!

Need to know about the **caller vs. callee calling conventions.**

Calling Conventions

- We've seen at least two options on how to implement function calls:
  - Use $a0 - $a3 , $v0 and $v1, and so on.
  - Push on stack
- There are many other variants.
  - For example, should caller or callee pop variables?
  - Or using registers instead of stack.
- These are called calling conventions.

Calling Conventions

A function can be both a caller and a callee (e.g., recursion).

- Caller vs. Callee
  - Caller is the function calling another function.
  - Callee is the function being called.
- We separate registers into:
  - Caller-Saved registers (`$t0-$t9`)
    - Also called "unsaved (or temporary) registers".
  - Callee-Saved registers (`$s0-$s7`)
    - Also called "saved registers"

Register Saving Conventions

Push them to the stack just before you call another function and restore them immediately after.

- Caller-Saved registers
  - Registers `8-15`, `24-25` (`$t0-$t9`): temporaries
  - Registers that the caller should save to the stack before calling a function. If they don't save them, there is no guarantee the contents of these registers will not be clobbered.
- Callee-Saved registers
  - Registers `16-23` (`$s0-$s7`): saved temporaries
  - It is the responsibility of the callee to save these registers and later restore them, if it's going to modify them.
  - Push them to the stack first thing in your function body and restore them just before you return!

Caller-Saved ($t0-$t9) vs. Callee-Saved ($s0-$s7) Registers

- Caller code
  - Using $to-$t9 and you care for their values?
    - Push them to the stack just before you make a function call and restore them immediately after the calling site.
    - If you don't care about the value, no need to do anything.
  - Using $so-$s7?
    - No action needed. It is the responsibility of the callee to ensure these registers are not modified.
- Callee code
  - Using $to-$t9?
    - No action needed. It it the responsibility of the caller to ensure there registers are not modified.
  - Using $so-s7?
    - You need to ensure these registers are not modified.
    - If you plan to modify them, push them to the stack in the beginning of your function and restore them in the very end just before the jr $ra.

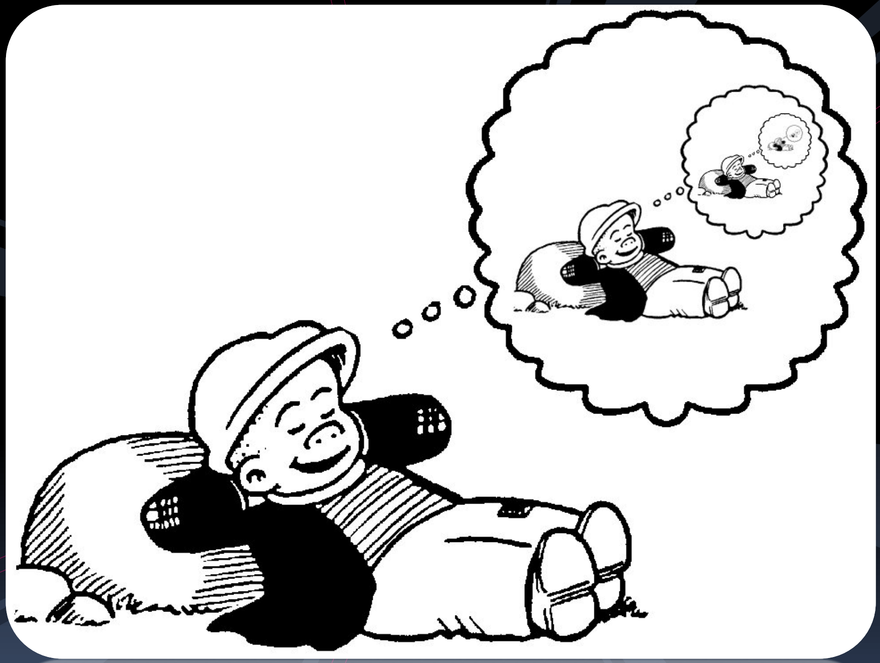
If a function is both a caller and a callee, it will fall under both categories.

Register Saving Conventions

- THESE ARE ONLY CONVENTIONS!!!!
  - There's nothing to enforce these rules
  - Not everyone actually agrees on what the convention should be
  - Not everyone follows the rules
  - Don't assume convention is being followed unless you're explicitly told
  - If in doubt, save it to the stack

Break

Recursion in Assembly



Example: factorial(int n)

Basic pseudocode for recursive factorial:

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in “product”
- Multiply product by n
  - Store in “result”
- Return result

Recursive programs

- How do we handle recursive programs?
  - Still needs base case and recursive step, as with other languages.

```
int factorial (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

  - Main difference: function is both caller and callee
  - So what?
  - Just make sure to preserve $ra and saved registers as we said

Recursive programs

- Solution: the stack!
  - Before recursive call, store the register values that you use onto the stack, and restore them when you come back to that point.
  - Don't forget to store $ra as one of those values, or else the program will loop forever!

```
int factorial (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

Factorial solution

- Steps to perform:
  - Load `x` from the stack.
  - Check if `x` is zero:
    - If `x==0`, push `1` onto the stack and return to the calling program.
    - If `x!=0`, push `x-1` onto the stack and call factorial again (i.e. jump to the beginning of the code).
    - After recursive call, pop result off of stack and multiply that value by `x`.
    - Push result onto stack, and return to calling program.

```
int fact (int x) {
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

factorial(int n)

- Load n off the stack
  - Store in `$t0`
- If `$t0 == 0`,
  - Push 1 onto stack
  - Return to caller
- If `$t0 != 0`,
  - Calculate n-1
  - Store and `$ra` onto stack
  - Push n-1 on `$t0` to stack
  - Call factorial

    - …time passes…

  - Pop the result of factorial (n-1) from stack, store in `$t2`
    - Also shrink stack (pop argument)
  - Restore `$ra` and `$t0` from stack
  - Multiply factorial (n-1) and n
  - Pop result onto stack
  - Return to calling program

- Base Case (n == 0)
  - return 1
- Get factorial(n-1)
  - Store result in "product"
- Multiply product by n
  - Store in "result"
- Return result

n → $to
n-1 → $t1
fact(n-1) → $t2

Translated recursive program (part 1)

```
main:        addi   $t0, $zero, 10
             addi   $sp, $sp, -4
             sw     $t0, 0($sp)
             jal    factorial
             ...
```

```
factorial:   lw     $t0, 0($sp)           # get x from stack
             bne    $t0, $zero, rec       # base case?
base:        addi   $t1, $zero, 1         # put return value
             addi   $sp, $sp, -4          #   onto stack
             sw     $t1, 0($sp)           #
             jr     $ra                   # return to caller
rec:         addi   $t1, $t0, -1          # x--
             addi   $sp, $sp, -4          # save $ra value
             sw     $ra, 0($sp)           #   onto stack
             addi   $sp, $sp, -4          # put x-1 on stack
             sw     $t1, 0($sp)           #   for rec call
             jal    factorial             # recursive call
```

Translated recursive program (part 2)

```
(continued from part 1 – returning from recursive call)
             lw     $t2, 0($sp)          # get return value
             addi   $sp, $sp, 8          #   from stack
             lw     $ra, 0($sp)          # restore return
             addi   $sp, $sp, 4          #   address value
             lw     $t0, 0($sp)          # restore x value
                                         #   for this call
             mult   $t0, $t2             # x*fact(x-1)
             mflo   $v0                  # fetch product
             addi   $sp, $sp, -4         # push n! result
             sw     $v0, 0($sp)          #   onto stack
             jr     $ra                  # return to caller
```

- Remember: `jal` always stores the next address location into `$ra`, and `jr` returns to that address.

Factorial stack view

The Stack Frame

Local Variables

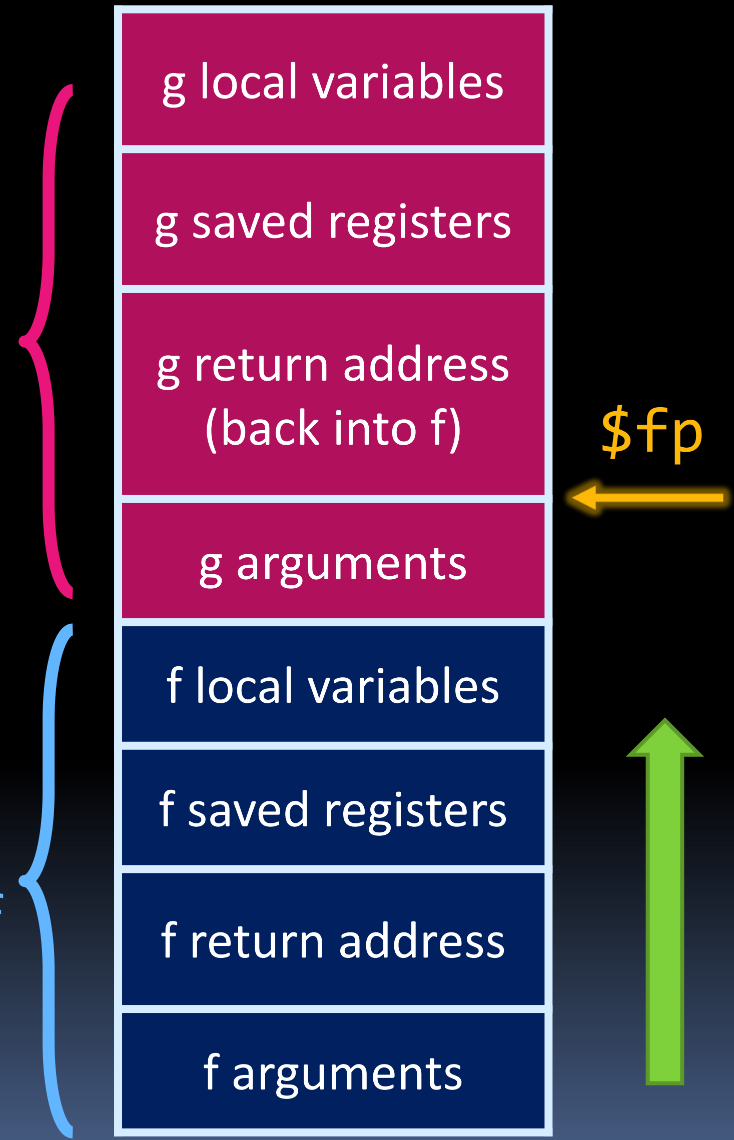
- Sometimes you just need local variables
  - You ran out of registers.
  - Or you want a local array.
  - You are compiling C code and the programmer is using many local variables.
- Local variables are local to the function.
- Where should I put them? On the stack!
  - Say the function needs 24 bytes for local variables
  - Just do `addi $sp,$sp,-24`
  - Before returning, restore `$sp` to how it was: `addi $sp,$sp,24`

The Stack Frame

- The stack frame is an area on the stack dedicated to each function.
- On the stack frame we store:
  - Arguments (stored by the caller)
  - Saved return address
  - Callee-saved registers (`$s0-$s7, $fp`)
  - Local variables
  - Caller-saved registers (`$t0-$t9`)
- Frame pointer `$fp` helps the function know what is where since we modify `$sp`.
  - Functions execute `add $fp, $zero, $sp` at entry

The Stack Frame

- Example:
  - main called f
  - f called g
- At entry:
  - push $ra
  - Push $fp
  - add $fp, $zero, $sp
- To return:
  - lw $ra, 0($fp)
  - Restore $sp
  - jr $ra



Review: Some Optimizations

- We started with always using the stack.
  - Do this unless we tell you otherwise!
- Changing the calling convention allows some nice optimizations:
  - Use saved registers wisely.
  - Pass arguments and return values in registers.
  - Keep arguments on stack, don't pop.
- Compilers can do even more:
  - Convert recursive calls to loops.
  - "Inlining" functions: move callee code into caller.

Almost Done!

- Left overs:
  - Interrupts
  - System calls
  - Odds and ends.
  - More dank memes.