Week 10, part D: The stack frame

Back To max3

```
def max3(a,b,c):
    tmp = max(a, b)
    res = max(tmp, c)
    return res
```

- 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)

Back To max3

```
def max3(a,b,c):
    tmp = max(a, b)
    res = max(tmp, c)
    return res
```

- 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)

wasteful: why pop $t2 just for pushing?

Avoiding Pop, Push, Pop

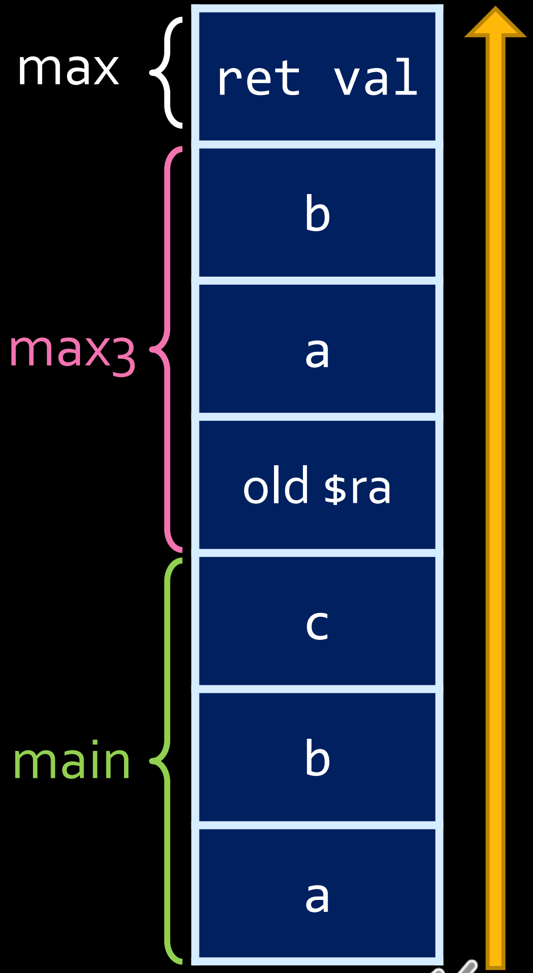
- Once we pop **c** off the stack into `$t2` we have to preserve it or it's gone forever.
- This constant push/pop is slow and wasteful.
    - Worse in more complex functions.
- What can we do?
- We can try using a saved register like `$s0`
    - What if we ran out of registers?
- Can we do better?

WHAT IF I TOLD YOU...

TO JUST LEAVE ARGUMENTS ON THE STACK?

Arguments Without Popping

- Let's try a new calling convention
- Caller will save any $to-$t9 registers
- Caller will save arguments on stack
- Callee will load arguments but not pop!
  - Access the memory, but will not modify $sp
- Callee computes, call other functions, etc.
- Callee pushes return value on stack
- Callee returns to caller
- Caller pops return value
- Caller pops arguments (shrinks stack)
- Caller restores registers



max3(a,b,c) with callee pop

- 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)

max3(a,b,c) with caller pop

- Load (not pop!) a, b (not c!) into registers $t0, $t1
- Push $ra
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into $t3
    - Also clear a,b from stack.
- Load c into $t2
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4 (and clear a,b)
- Pop $ra
- Push $t4 final max
- Return to caller (jr $ra)

By keeping things on stack we avoid useless work needed to save and restore

Reflection

- We created an alternative “caller pop” calling convention:
  - Caller responsible for pushing and popping arguments.
  - Callee responsible for pushing and popping saved registers.
- It’s almost as if each function has an area of the stack dedicated to its use.
- What else can we do with it?

Local Variables

```
int func(int a, int b) {
    int local_array[256];
    ...
}
```

- Sometimes we just need local variables
  - We ran out of registers to hold variables.
  - Or we want a local array or local structs.
  - 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`

Example

- Use the new caller-pop convention to implement:

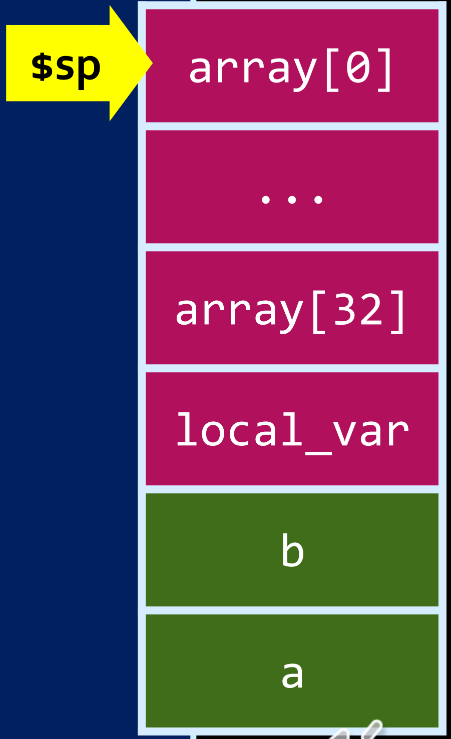
```
int func(int a, int b) {
    int local_array[33];
    int local_var = 4;
    local_array[0] = a;
    local_array[20] = -55;
    return b + local_var;
}
```

```
int func(int a, int b) {
    int local_array[33];
    int local_var = 4;
    local_array[0] = a;
    local_array[20] = -55;
    return b + local_var;
}
```

```
func:   # we'll use the caller-pop convention
        # load a into $t0, b into $t1
        lw $t0, 4($sp)
        lw $t1, 0($sp)
        # make space for array local_var
        # we need 136 bytes in total: 33*4 (array) + 4 (var)
        addi $sp, $sp, -136
        # array base is at $sp, local_var address is $sp+132
```

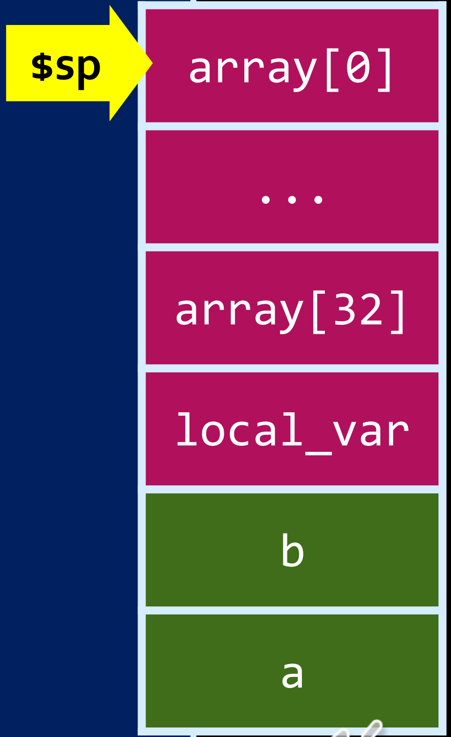
```c
int func(int a, int b) {
    int local_array[33];
    int local_var = 4;
    local_array[0] = a;
    local_array[20] = -55;
    return b + local_var;
}
```

```
func:   # we'll use the caller-pop convention
        # load a into $t0, b into $t1
        lw $t0, 4($sp)
        lw $t1, 0($sp)
        # make space for array local_var
        # we need 136 bytes in total: 33*4 (array) + 4 (var)
        addi $sp, $sp, -136
        # array base is at $sp, local_var address is $sp+132
```



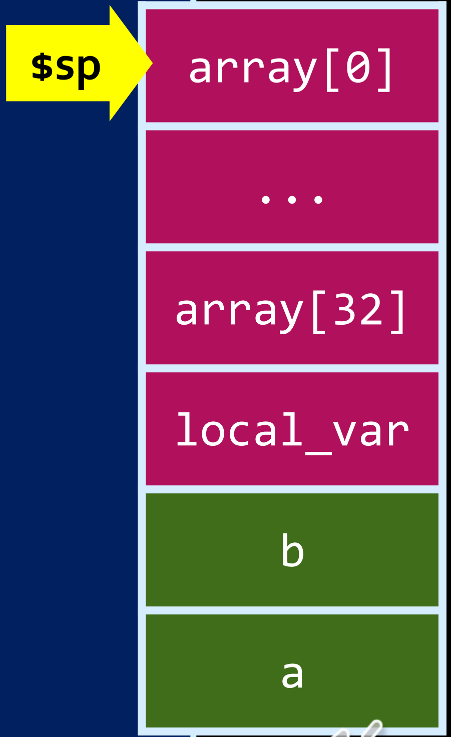
```
int func(int a, int b) {
    int local_array[33];
    int local_var = 4;
    local_array[0] = a;
    local_array[20] = -55;
    return b + local_var;
}
```

```
func:   # we'll use the caller-pop convention
        # load a into $t0, b into $t1
        lw $t0, 4($sp)
        lw $t1, 0($sp)
        # make space for array local_var
        # we need 136 bytes in total: 33*4 (array) + 4 (var)
        addi $sp, $sp, -136
        # array base is at $sp, local_var address is $sp+132
        # set local_var = 4
        li $t2, 4
        sw $t2, 132($sp)
        # local_array[0] = a
        sw $t0, 0($sp)
        # local_array[20] = -55
        li $t2, -55
        sw $t2, 80($sp)            # address is $sp + 20*4
```



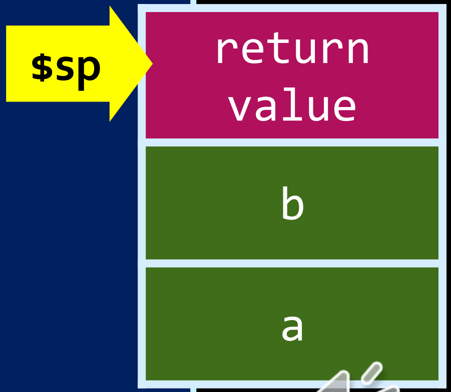
```
int func(int a, int b) {
    int local_array[33];
    int local_var = 4;
    local_array[0] = a;
    local_array[20] = -55;
    return b + local_var;
}
```

```
func:   # we'll use the caller-pop convention
        # load a into $t0, b into $t1
        lw $t0, 4($sp)
        lw $t1, 0($sp)
        # make space for array local_var
        # we need 136 bytes in total: 33*4 (array) + 4 (var)
        addi $sp, $sp, -136
        # array base is at $sp, local_var address is $sp+132
        # set local_var = 4
        li $t2, 4
        sw $t2, 132($sp)
        # local_array[0] = a
        sw $t0, 0($sp)
        # local_array[20] = -55
        li $t2, -55
        sw $t2, 80($sp)          # address is $sp + 20*4
        # compute b + local_var
        lw $t2, 132($sp)
        add $t1, $t1, $t2 # $t1 = b + local_var
```



```
int func(int a, int b) {
    int local_array[33];
    int local_var = 4;
    local_array[0] = a;
    local_array[20] = -55;
    return b + local_var;
}
```

```
func:   # we'll use the caller-pop convention
        # load a into $t0, b into $t1
        lw $t0, 4($sp)
        lw $t1, 0($sp)
        # make space for array local_var
        # we need 136 bytes in total: 33*4 (array) + 4 (var)
        addi $sp, $sp, -136
        # array base is at $sp, local_var address is $sp+132
        # set local_var = 4
        li $t2, 4
        sw $t2, 132($sp)
        # local_array[0] = a
        sw $t0, 0($sp)
        # local_array[20] = -55
        li $t2, -55
        sw $t2, 80($sp)         # address is $sp + 20*4
        # compute b + local_var
        lw $t2, 132($sp)
        add $t1, $t1, $t2 # $t1 = b + local_var
        # clean up stack
        addi $sp, $sp, 136
        # push return value
        addi $sp, $sp, -4
        sw $t1, 0($sp)
        jr $ra
```

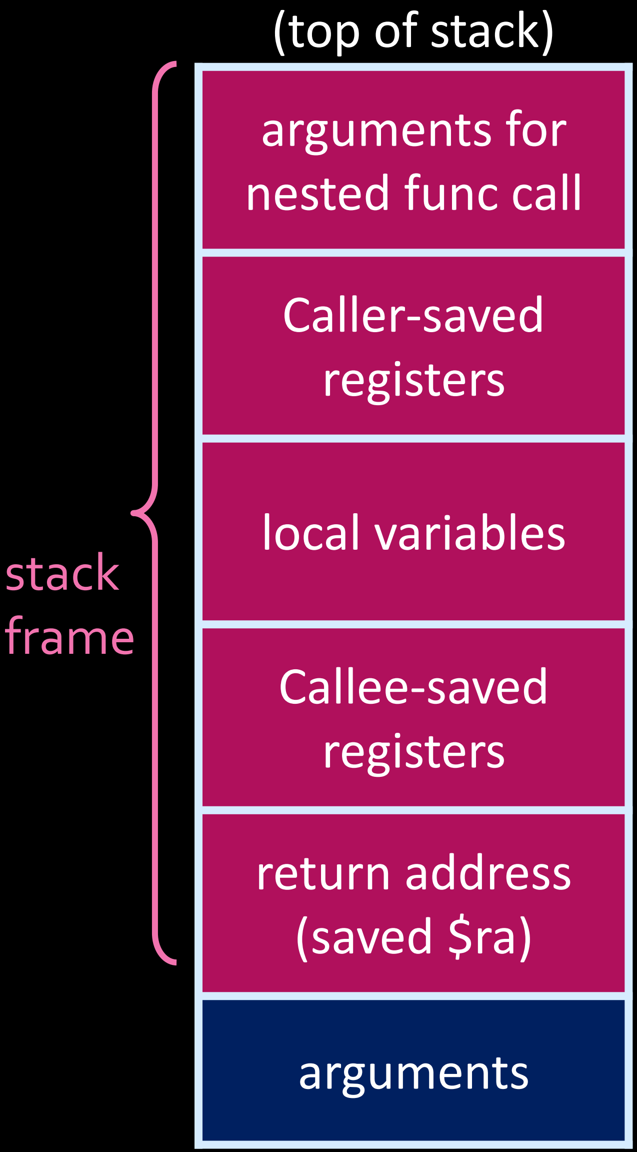


The Stack Frame

- Stack frame: a space on the stack that a function allocates for itself.
  - The function is responsible for setting it up and cleaning after itself.
- On the stack frame we store:
  - Saved return address
  - Callee-saved registers (`$s0-$s7, $fp`)
  - Local variables.
  - Caller-saved registers (`$t0-$t9`)

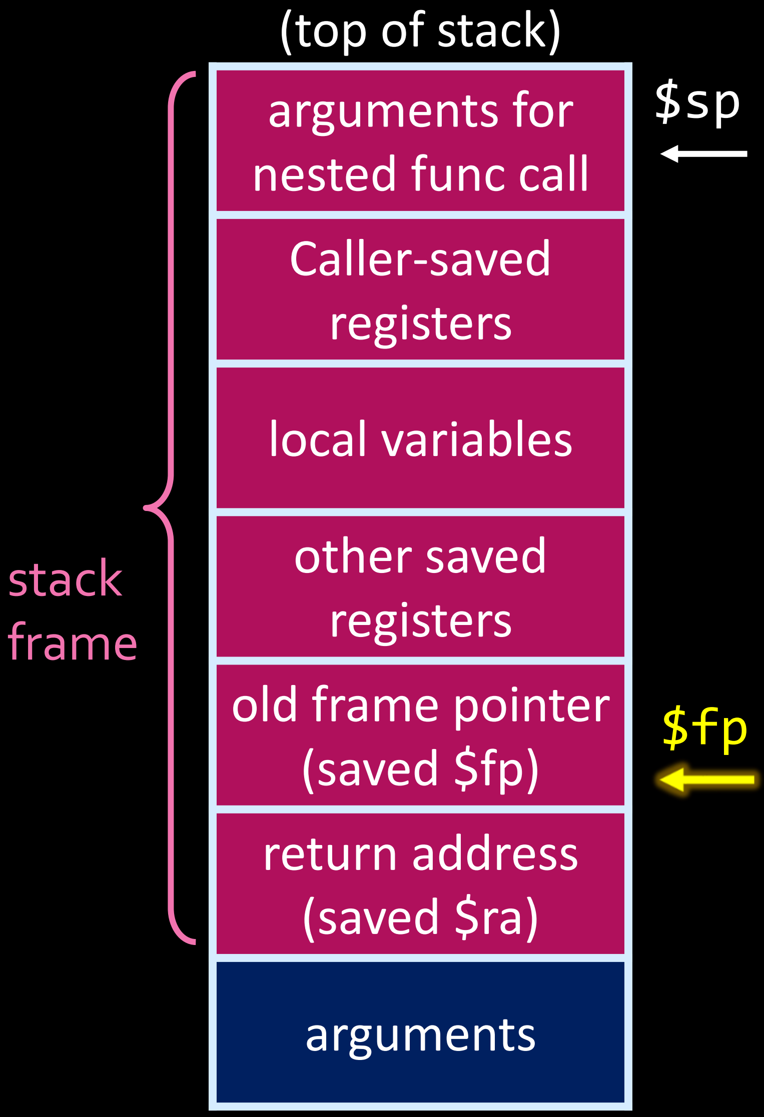
The Stack Frame

- Stack frame: a space on the stack that a function allocates for itself.
  - The function is responsible for setting it up and cleaning after itself.
- On the stack frame we store:
  - Saved return address
  - Callee-saved registers (`$s0-$s7, $fp`)
  - Local variables.
  - Caller-saved registers (`$t0-$t9`)
- Structure determined by the calling convention



The Stack Frame

- We store arguments, variables and more on the stack frame.
- But we often need to change $sp (for push/pop).
- How do we find what we need?
- Use the frame pointer $fp to point to the start of the stack frame:
  - At entry, functions save $sp to $fp
  - Modifying $sp won't affect $fp.
  - Must save old $fp too.



(top of stack)

$sp

stack frame

arguments for nested func call

Caller-saved registers

local variables

other saved registers

old frame pointer (saved $fp)

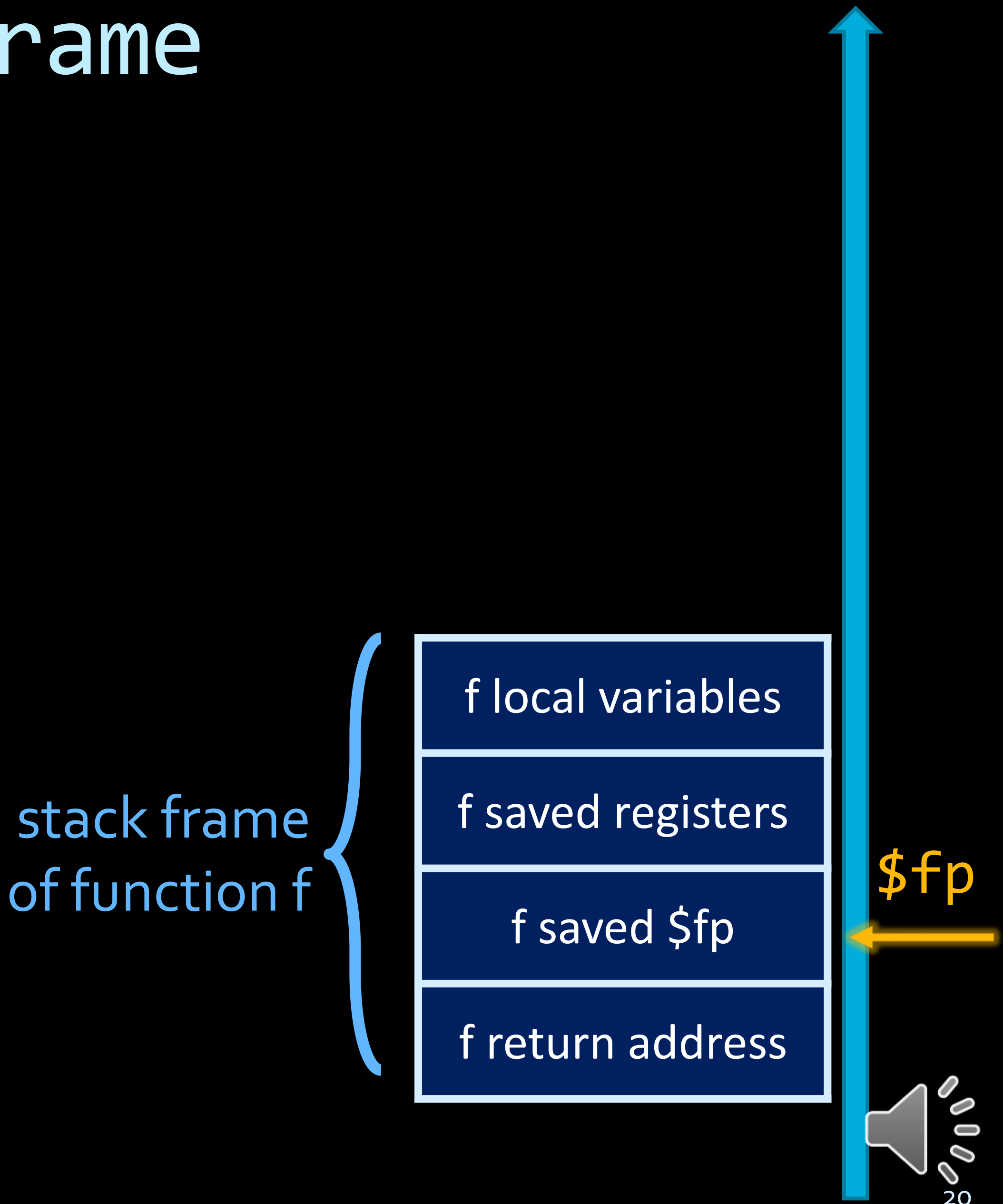
$fp

return address (saved $ra)

arguments

The Stack Frame

- Example:
  - main called f
  - f calling g



stack frame of function f

f local variables

f saved registers

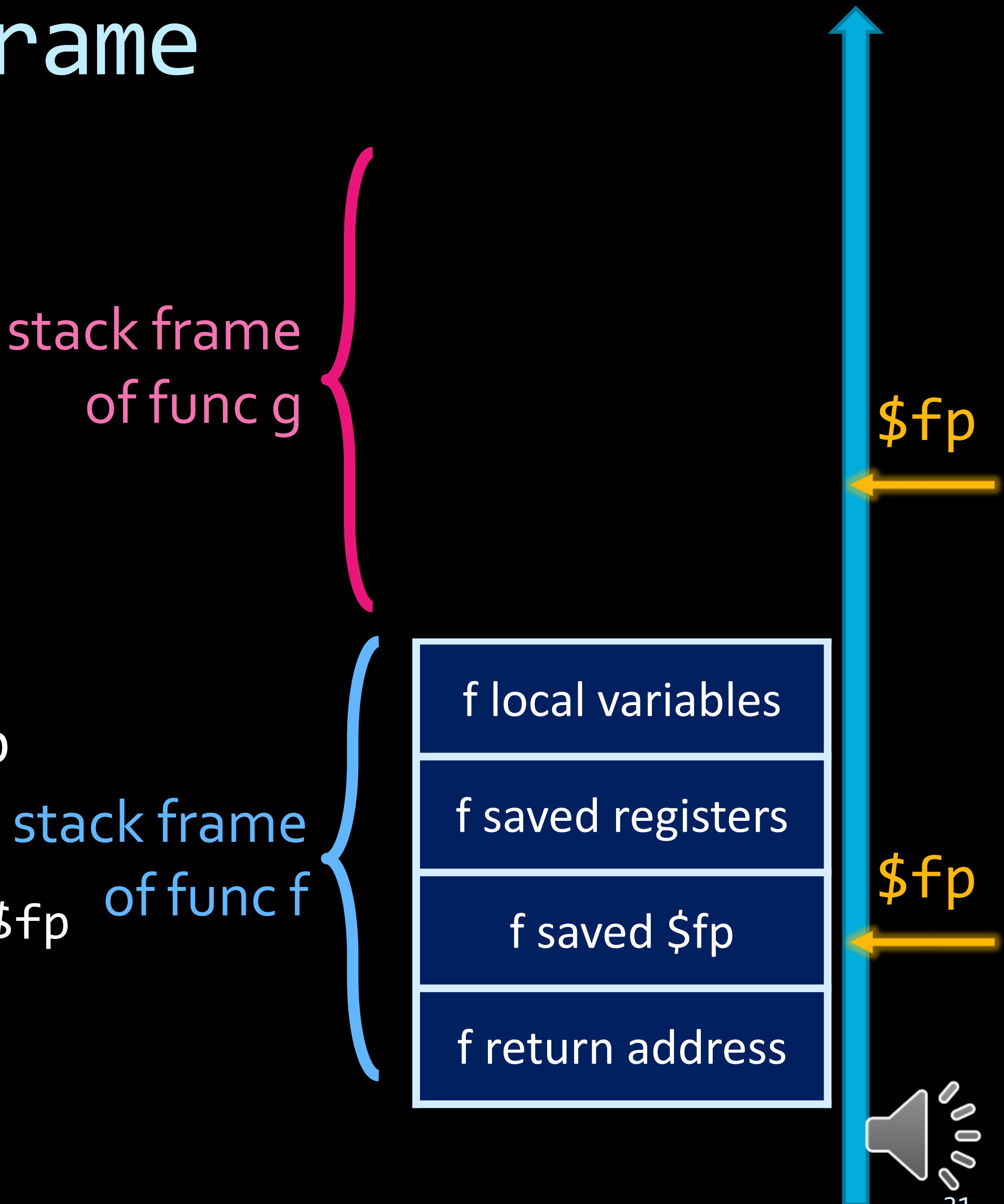
f saved $fp

f return address

$fp

The Stack Frame

- Example:
  - main called f
  - f calling g
- At entry:
  - Push `$ra`
  - Push `$fp`
  - `add $fp, $zero, $sp`
- To return:
  - Restore $sp: `move $sp, $fp`
  - Pop $fp from stack
  - Pop $ra from stack
  - `jr $ra`



Stack-frame Calling convention

Caller (before)

- Push $t0 - $t9 if needed.
    - Also $a0-$a3, $v0-$v1
- Push arguments
    - Or put in $a0-$a3
- Call using jal

Callee (start)

- Push $ra
- Push $fp
- $fp = $sp
- Push $s0-$s7 (if needed)
- Make space for variables:
  $sp = $sp - size of local vars

Callee can now write to $a0-$a3, $v0-v1, $t0-t9, and any saved $s0-s7. Callee can also push and pop, and call functions.

Stack-frame Calling convention

Callee (end)

- Restore $s7-$s0 (reverse order)
- Restore $sp: $sp = $fp
- Pop $fp
- Pop $ra
- Push return value
  - or put in $v0-$v1
- Return to caller: jr $ra

Caller (after)

- Pop return value
  - If it's not in $v0-$v1
- Clear arguments from stack
- Pop $t9-t0

Advice for Stack Frames

- Any space you allocate on the stack, you should later de-allocate.
  - If you pushed it there, you have to pop it.
  - Function always leaves the stack the way it found it.
  - The only exception is return value.
- Remember to pop the items in reverse order.
  - It might help to draw a diagram of how your stack will look like.
- When pushing / popping more than one item:
  - Either allocate space as you go: addi $sp, $sp, -4
  - Either allocate all the space in one go

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.
  - Use this for the project!
- Compilers can do even more:
  - Convert recursive calls to loops.
  - “Inlining” functions: move callee code into caller.

Summary of Calling Functions

- Simple stack calling convention (use this):
  - Caller pushes arguments, callee pops them.
  - Callee pushes return values, caller pops them,
  - Save $ra if you have a nested / recursive call.
  - Save $t0-$t9 / $s0-s7 registers if you need to.
    - Based on the rules we defined before.
- Argument-based variant ($a0-$a3, $v0-$v1)
- Caller-pop variant
- Stack-frame

Almost Done!

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