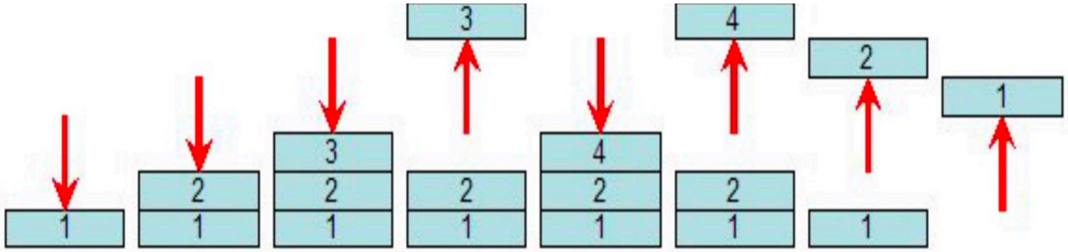


Using the Stack
MIPS Calling Convention for Functions

CS 64: Computer Organization and Design Logic

Lecture #10

Fall 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- NO LAB THIS WEEK!
- Your current lab (#5) is due **Friday 2/14**

What’s on the Midterm on **Wednesday**?

What’s on It?

- Everything we’ve done so far from start to Monday, 2/10
  - Except recursive functions

What Should I Bring?

- Your pencil(s), eraser, MIPS Reference Card (on 1 page)
- THAT’S ALL!

What Else Should I Do?

- ***IMPORTANT***: Come to the classroom 5-10 minutes EARLY
- **If you are late, I may not let you take the exam**
- ***IMPORTANT***: Use the bathroom before the exam – once inside, you cannot leave
- I will have some of you re-seated
- Bring your UCSB ID

Lecture Outline

- More on the MIPS Calling Convention
- Recursive Functions

Any Questions From Last Lecture?

The MIPS Convention In Its Essence

Preserved vs **Unpreserved** Regs

- **Preserved**: **$s0 - $s7**, and **$sp , $ra**
- **Unpreserved**: **$t0 - $t9**, **$a0 - $a3**, and **$v0 - $v1**

- Values held in **Preserved Regs** immediately before a function call ***MUST be the same*** immediately after the function returns.

- Values held in **Unpreserved Regs** must always be assumed to change after a function call is performed.
  - $a0 - $a3 are for passing arguments into a function
  - $v0 - $v1 are for passing values from a function

An Example…
*Consider when **functionX()** calls **functionY()***

functionX:

uses **`$s0, $s1, $s2, $s3`**

Then calls function

 # li $a0, 42

 # jal functionY

functionY:

DOES NOT know what **$s regs.** functionX used

DOES know what **$s regs. it** plans on using (say, **`$s0, $s1`** only)

Plan for functionY:

 # 1. PUSH $s0, $s1 only into stack

(i.e. saves them for functionX)

 # 2. Does its calcs/instructions

 # 3. POPS $s0, $s1 back from stack

(i.e. recovers them for functionX)

 # 4. Returns via jr $ra

MIPS Call Stack

- We know what a Stack is…
- A **"Call Stack"** is used for storing ***the return addresses*** of the various **functions** which have been *called*

- When you **call** a function (e.g. `jal funcA`), the address that we need to return to is **pushed** into the call stack.

…

***funcA* does its thing… then…**

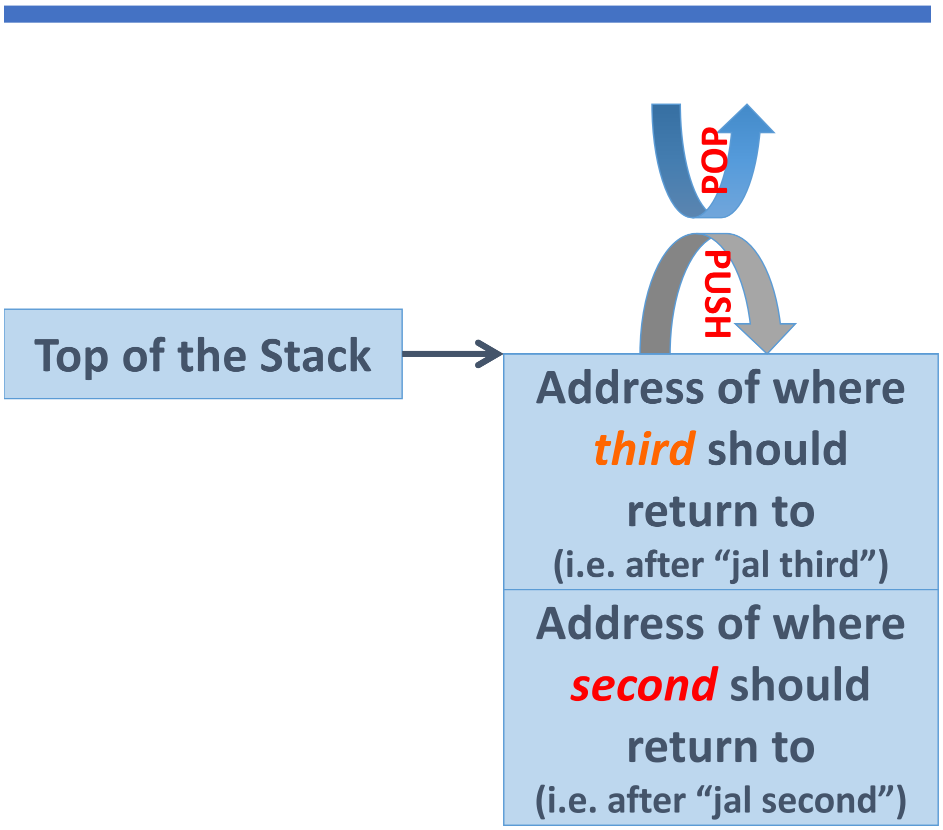
…

The function needs to return.

So, the address is **popped** off the call stack

```
void first()
{
   second()
   return; }

void second()
{
   third ();
   return; }

void third()
{
   fourth ();
   return; }

void forth()
{
   return; }
```

Stack



Top of the Stack

Address of where *third* should return to (i.e. after "jal third")

Address of where *second* should return to (i.e. after "jal second")

```
fourth:
   jr $ra

third:
   push $ra
   jal fourth
   pop $ra
   jr $ra

second:
   push $ra
   jal third
   pop $ra
   jr $ra

first:
   jal second

li $v0, 10
syscal
```

Why ***addiu?***
Because there is no such thing as a negative memory address
AND
we want to avoid triggering a processor-level ***exception on overflow***

```
fourth:
  jr $ra

third:
  addiu $sp, $sp, -4
  sw $ra, 0($sp)
  jal fourth
  lw $ra, 0($sp)
  addiu $sp, $sp, 4
  jr $ra

second:
  addiu $sp, $sp, -4
  sw $ra, 0($sp)
  jal third
  lw $ra, 0($sp)
  addiu $sp, $sp, 4
  jr $ra

first:
  jal second

li $v0, 10
  syscall
```

```
fourth:
  jr $ra

third:
  push $ra
  jal fourth
  pop $ra
  jr $ra

second:
  push $ra
  jal third
  pop $ra
  jr $ra

first:
  jal second

li $v0, 10
syscal
```

An Illustrative Example

```
…
…
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}


int doSomething(int x, int y)
{
  int a = subTwo(x, y);
  int b = subTwo(y, x);
  return a + b;
}
…
…
```

<u>subTwo doesn't call anything</u>

What should I map **a** and **b** to?

$a0 and *$a1*

Can I map **sub** to **$t0**?

*Ok, b/c I don't care about **$t****
(not the best tactic, tho…)

*Eventually, I have to have **sub** be **$v0***

<u>doSomething DOES call a function</u>

What should I map **x** and **y** to?

*Since we want to preserve them across the call to subTwo, we should map them to **$s0** and **$s1***

What should I map **a** and **b** to?

*"**a+b**" has to eventually be **$v0**. I should make at least **a** be a preserved reg (**$s2**). Since I get **b** back from a call and there's <u>no other call after it</u>, I can likely get away with not using a preserved reg for **b**.*

```
subTwo:
sub $v0, $a0, $a1
jr $ra

doSomething:
# preserve for the sake
# of whatever called
# doSomething
addiu $sp, $sp, -16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)

move $s0, $a0
move $s1, $a1

jal subTwo

move $s2, $v0
```

```
move $a0, $s1
move $a1, $s0

jal subTwo

add $v0, $v0, $s2

# pop back the preserved
# so that they're ready
# for whatever called
# doSomething
lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
lw $ra, 12($sp)
addiu $sp, $sp, 16

jr $ra
```

```
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

int doSomething(int x, int y)
{
  int a = subTwo(x, y);
  int b = subTwo(y, x);
  return a + b; }
```

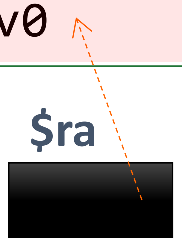
```
subTwo:
sub $v0, $a0, $a1
jr $ra

doSomething:
addiu $sp, $sp, -16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)

move $s0, $a0
move $s1, $a1

jal subTwo
move $s2, $v0
```

```
move $a0, $s1
move $a1, $s0

jal subTwo

add $v0, $v0, $s2

lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
lw $ra, 12($sp)
addiu $sp, $sp, 16

jr $ra
```

```
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

int doSomething(int x, int y)
{
  int a = subTwo(x, y);
  int b = subTwo(y, x);
  …
  return a + b;
}
```

stack

- Orig. $s0
- Orig. $s1
- Orig. $s2
- Orig. $ra



	$a0	$a1	
Arguments	*int x*	*int y*	
	$s0	$s1	$s2
Preserved	*int x*	*int y*	*x – y*
	$t0 - $t9		
Unpreserved			
	$v0		
Result Value	*a – b*		

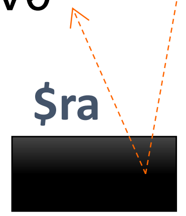
```
subTwo:
sub $v0, $a0, $a1
jr $ra

doSomething:
addiu $sp, $sp, -16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)

move $s0, $a0
move $s1, $a1

jal subTwo
move $s2, $v0
```

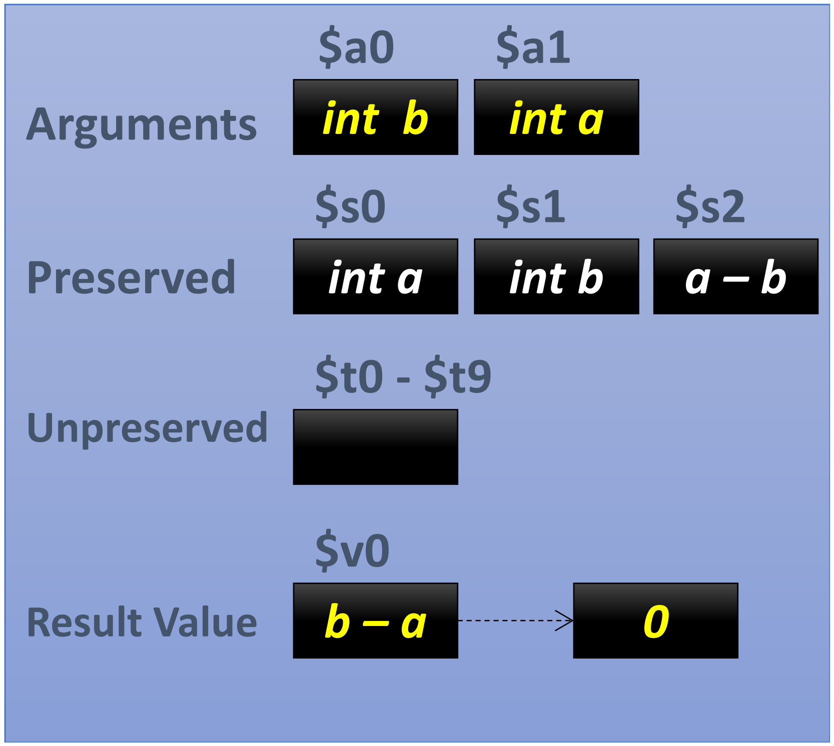
```
move $a0, $s1
move $a1, $s0

jal subTwo

add $v0, $v0, $s2

lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
lw $ra, 12($sp)
addiu $sp, $sp, 16

jr $ra
```

```
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

int doSomething(int x, int y)
{
  int a = subTwo(x, y);
  int b = subTwo(y, x);
  …
  return a + b;
}
```

stack

- **Orig. $s0**
- **Orig. $s1**
- **Orig. $s2**
- **Orig. $ra**





Arguments: $a0 ***int b***, $a1 ***int a***

Preserved: $s0 ***int a***, $s1 ***int b***, $s2 ***a – b***

Unpreserved: $t0 - $t9

Result Value: $v0 ***b – a*** → ***0***

```
subTwo:
sub $v0, $a0, $a1
jr $ra

doSomething:
addiu $sp, $sp, -16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)

move $s0, $a0
move $s1, $a1

jal subTwo
move $s2, $v0
```

```
move $a0, $s1
move $a1, $s0

jal subTwo

add $v0, $v0, $s2
```

```
lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
lw $ra, 12($sp)
addiu $sp, $sp, 16

jr $ra
```

```
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

int doSomething(int x, int y)
{
  int a = subTwo(x, y);
  int b = subTwo(y, x);
  …
  return a + b;
}
```

stack

Orig. $s0

Orig. $s1

Orig. $s2

Orig. $ra

$ra

Original caller $ra

	$a0	$a1	
Arguments	*int b*	*int a*	
	$s0	$s1	$s2
Preserved	*orig.*	*orig.*	*orig.*
	$t0		
Unpreserved			
	$v0		
Result Value	*0*		

Lessons Learned re: MIPS C.C.

- We pass arguments into the functions using **`$a*`**
- If we use **`$s*`** to work out calculations in registers ***then we should preserve their old values FIRST***, so we make sure to save them in the call stack
  - These are var values that DO need to live beyond a call
  - In the end, the original values were returned back
- We *could* use **`$t*`** to work out some calcs. in regs ***that we did not need to preserve***
  - These values DO NOT need to live beyond a function call
- We use **`$v*`** as regs. to return the value of the function

An Example Using Recursion

```
int add(int n): {
        if (n < 0) return 0;          // base case
        return n + add(n-1);          // recursive case
}

int main(): {
        int n = 5;
        cout << add(n);
}
```

I expect:

5 + 4 + 3 + 2 + 1 = 15

```
 .text
# $a0: n
# $v0: result
# sum(n) = sum of all numbers from 0 to n
sum:
    addiu $sp, $sp, -8     # PUSH
    sw $ra, 4($sp)
    sw $s0, 0($sp)

    li $v0, 0                   # Initialize sum ($v0)
    blt $a0, $zero, return      # If size < 0, then you've reached the base case

    move $s0, $a0       # preserve a0 (variable n)
    addi $a0, $a0, -1   # n is now: n - 1
    jal sum             # recursive call

return:
    add $v0, $v0, $s0   # add n to $v0

    lw $ra, 4($sp)      # POP
    lw $s0, 0($sp)
    addiu $sp, $sp, 8

    jr $ra

main:
    li $a0, 5       # n = 5 (arbitrary, for this example)
    jal sum         # call function sum(4), return answer in $v0

exit:
    move $a0, $v0
    li $v0, 1
    syscall

    li $v0, 10
    syscall
```

Recursive Functions

- This same setup handles nested function calls and recursion
  - i.e. By saving **$ra** methodically on the stack (and any **$s** regs that need it too…)

- Example: recursive_fibonacci.asm

recursive_fibonacci.asm

Recall the Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, etc…

$$fib(n) = fib(n-1) + fib(n-2)$$

In C/C++, we might write the recursive function as:

Base cases

```
int fib(int n)
{
      if (n == 0)
            return (0);
      else
            if (n == 1)
                  return (1);
            else
                  return (fib(n-1) + fib(n-2));
}
```

recursive_fibonacci.asm

- We’ll need at least 3 registers to keep track of:
  - The (single) input to the call, i.e. var **n**
    (it changes with each recursion and we’ll need it when tallying up)
  - The output or partial output to the call *(same reason)*
  - The value of **$ra** (since this is a recursive function)

- We’ll use $s* registers b/c **we need to preserve these vars/regs. beyond the function call**

If we make $s0 = **n** and $s1 = **fib(n – 1)**

- Then we need to save $s0, $s1 and $ra on the stack in the “fibonnaci” function
  - So that we do not corrupt/lose what’s already in these regs

recursive_fibonacci.asm

- So, we start off in the **`main:`** portion
  - **n** is our argument into the function, so it's in **$a0**

- We'll put our number (example: 7) in $a0 and then call the function "fibonacci"
  - i.e.

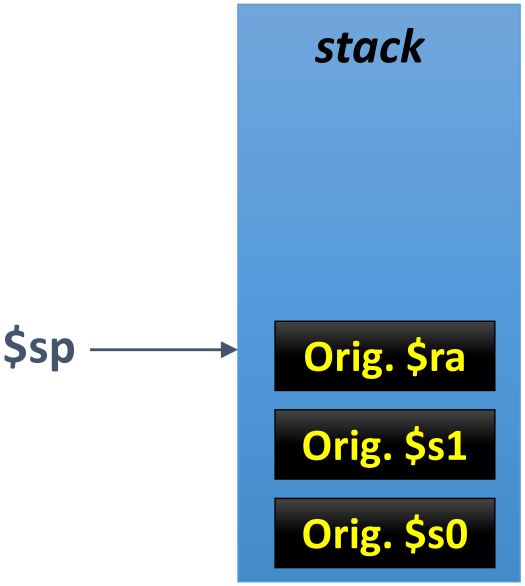
```
li $a0, 7
jal fibonacci
```

recursive_fibonacci.asm
Inside the function "fibonacci"

- **First**: Check for the base cases
  - Is **n** (`$a0`) equal to 0 or 1?
  - Branch accordingly

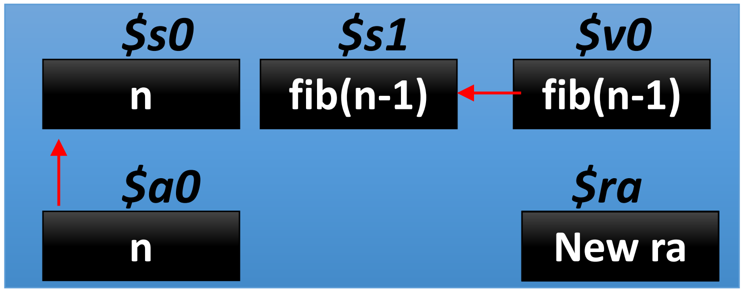
- Next: Do the recursion --- but first…!

  We need to plan for 3 words in the stack
  - **$sp = $sp – 12**
  - **Push** 3 words in (i.e. 12 bytes)
  - The order by which you put them in does *not strictly* matter, *but* it makes more "organized" sense to ***push $s0, then $s1, then $ra***

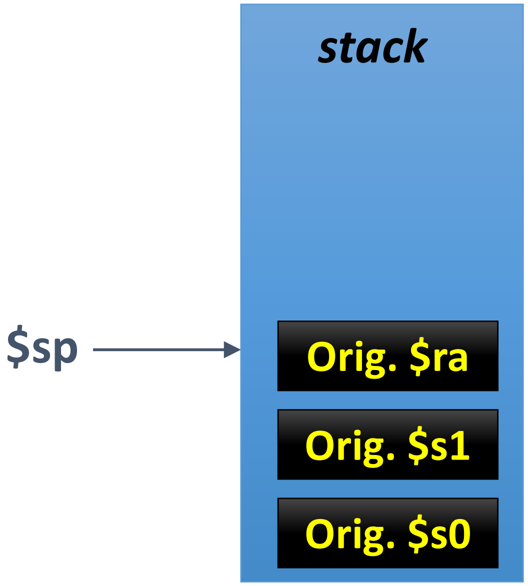


recursive_fibonacci.asm

- Next: calculate fib(n – 1)
  - Call recursively & copy output ($v0) in $s1
- Next: calculate fib(n – 2)



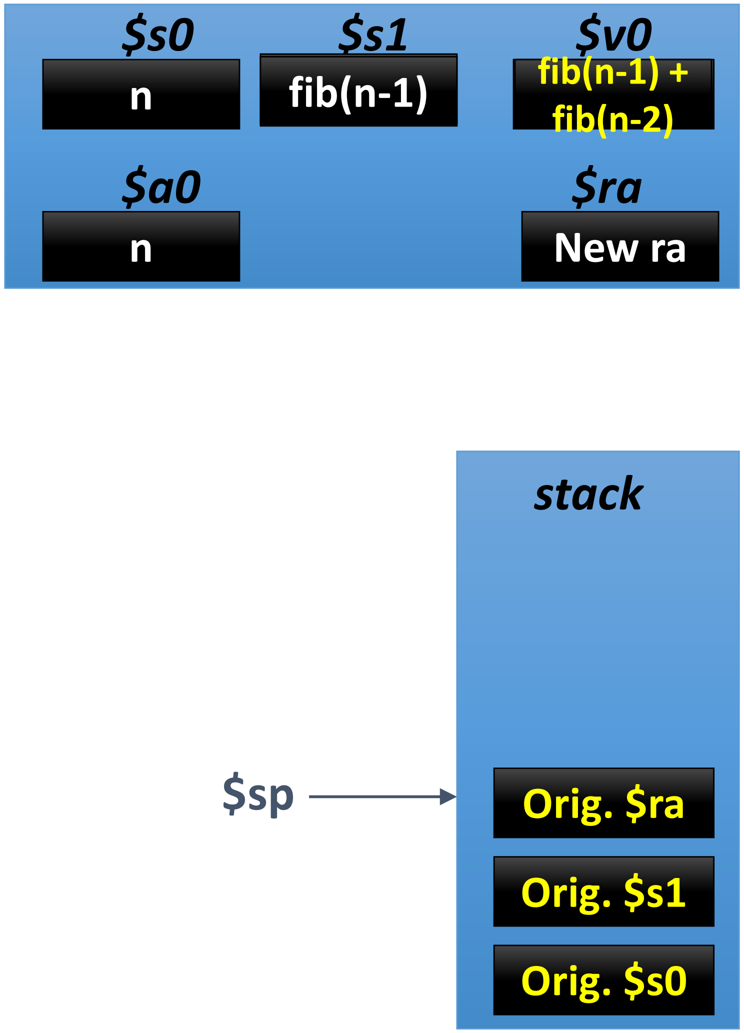
$s0
n
$s1
fib(n-1)
$v0
fib(n-1)
$a0
n
$ra
New ra



stack
$sp
Orig. $ra
Orig. $s1
Orig. $s0

recursive_fibonacci.asm

- Next: calculate fib(n – 1)
  - Call recursively & copy output ($v0) in $s1
- Next: calculate fib(n – 2)
  - Call recursively & add $s1 to the output ($v0)



$s0
n

$s1
fib(n-1)

$v0
fib(n-1) + fib(n-2)

$a0
n

$ra
New ra

stack

$sp

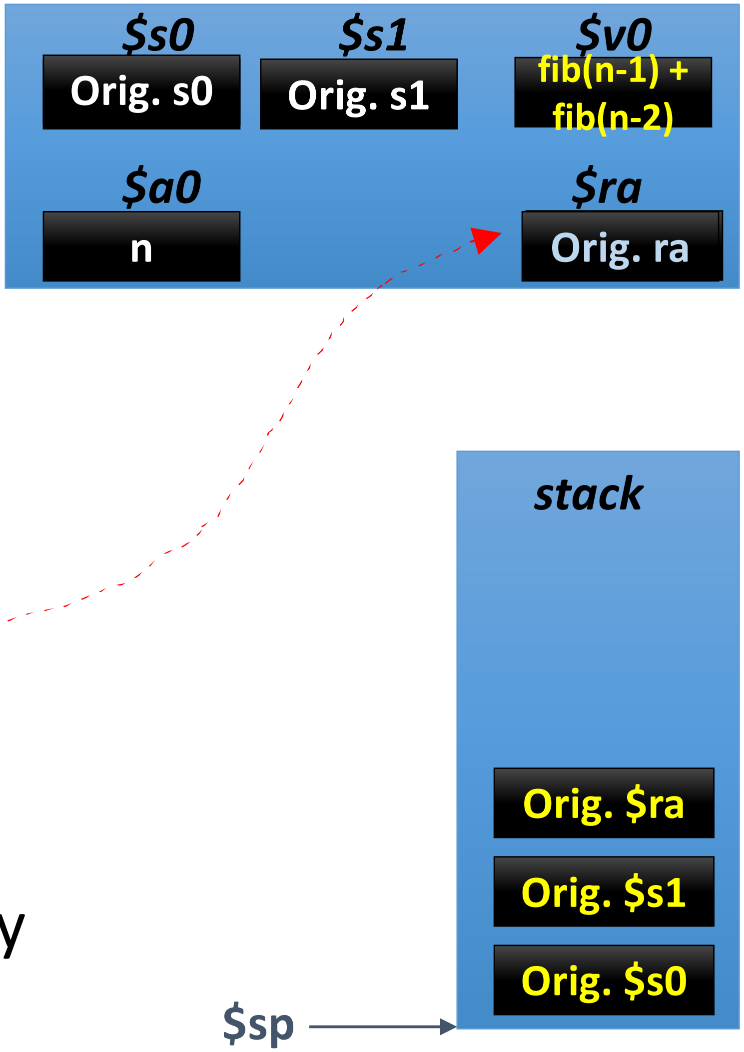
Orig. $ra

Orig. $s1

Orig. $s0

recursive_fibonacci.asm

- Next: calculate fib(n – 1)
  - Call recursively & copy output ($v0) in `$s1`
- Next: calculate fib(n – 2)
  - Call recursively & add `$s1` to the output ($v0)
- Next: restore registers
  - Pop the 3 words back to `$s0`, `$s1`, and `$ra`
- Next: return to caller (i.e. main)
  - Issue a **`jr $ra`** instruction



- Note how when we leave the function and go back to the "callee" (main), we did not disturb what was in the registers previously
- And now we have our output where it should be, in **`$v0`**

A Closer Look at the Code

- Open **recursive_fibonacci.asm**

Tail Recursion

- Check out the demo file **tail_recursive_factorial.asm** at home
- What's special about the *tail recursive functions* (see example)?
  - **Where the recursive call is the very last thing in the function.**
  - With the right optimization, it can **use a constant stack space (no need to keep saving $ra over and over – it's more efficient)**

```
int TRFac(int n, int accum)
{
    if (n == 0)
        return accum;
    else
        return TRFac(n - 1, n * accum)
}
```

For example, if you said:
TRFac(4, 1)

Then the program would **return**:
TRFac(3, 4), then return
TRFac(2, 12), then return
TRFac(1, 24), then return
TRFac(0, 24), then, since **n = 0**,
It would return 24

Your To-Dos

- **STUDY FOR THE MIDTERM EXAM!**

- Go over the **fibonnaci.asm** and **tail_recursive_factorial.asm** programs

- Work on Assignment #5
  - Due on **Friday**

</LECTURE>