# Function Calling Conventions 2

**CS 64: Computer Organization and Design Logic**
**Lecture #10**

Ziad Matni
Dept. of Computer Science, UCSB

# Lecture Outline

- More on MIPS Calling Convention
  - Functions calling functions
  - Recursive functions

# Administrative

- Re: Midterm Exam #1

# Administrative

- Mid-quarter evaluations for T.As and for Prof.
  - Links on the last slide and will put up on Piazza too
  - Optional to do, but very appreciated by us all!

# Any Questions From Last Lecture?

# Is-Not-A-Quiz-But-You-Should-Think-About-It

***Consider this C/C++ code:***

```
void third() {}
void second() {third();}
void first() {second();}
int main() {first();}
```

***And consider this supposedly equivalent MIPS code*** ➔ ➔

a) Are there any errors in it? (i.e. will it run?)

b) Does it follow the MIPS C.C.? **EXPLAIN YOUR ANSWER**

```
third:
   jr $ra
second:
   move $t0, $ra
   jal third
   jr $t0
first:
   move $t1, $ra
   jal second
   jr $t1
main:
   jal first
   li $v0, 10
   syscall
```

# Lecture Outline

- Recapping MIPS Calling Convention
  - Function calling function example
  - Recursive function example

# The MIPS Convention In Its Essence

- Remember: Preserved vs Unpreserved Regs
- **Preserved**: **$s0 - $s7**, and **$sp**
- **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 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;
}
```

**subTwo doesn't call anything**

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

*$a0 and $a1*

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

*Yes, b/c I don't care about $t**

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

**doSomething DOES call a function**

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 no other call after it, I can likely get away with not using a preserved reg for **b**.*

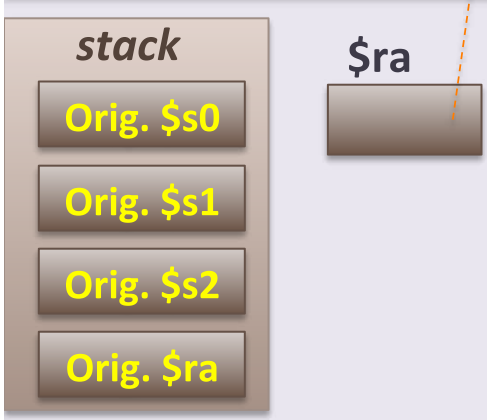
```
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

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

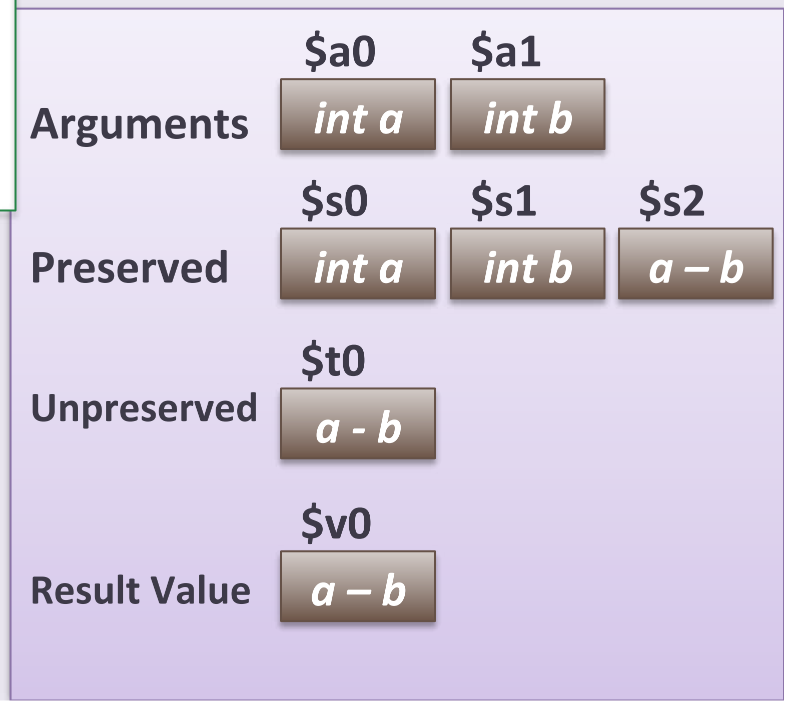
```
subTwo:
sub $t0, $a0, $a1
move $v0, $t0
jr $ra

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

jr $ra
```

```
subTwo:
sub $t0, $a0, $a1
move $v0, $t0
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 $ra, 12($sp)
lw $s2, 8($sp)
lw $s1, 4($sp)
lw $s0, 0($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 $t0, $a0, $a1
move $v0, $t0
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 $ra, 12($sp)
lw $s2, 8($sp)
lw $s1, 4($sp)
lw $s0, 0($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

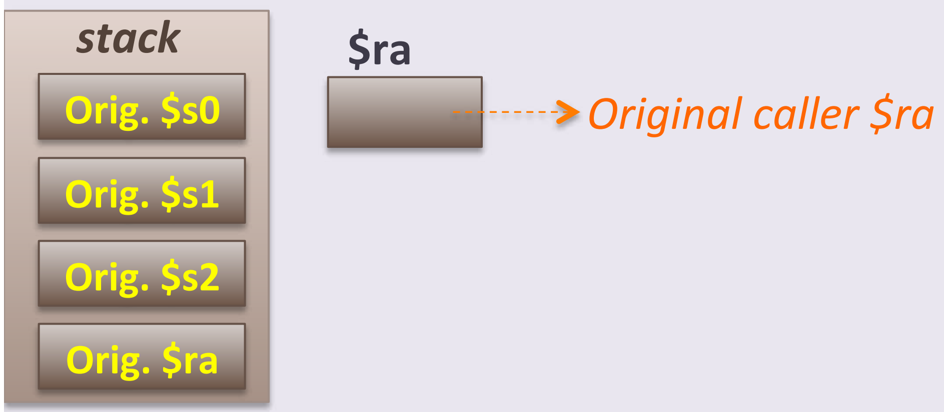
Orig. $ra

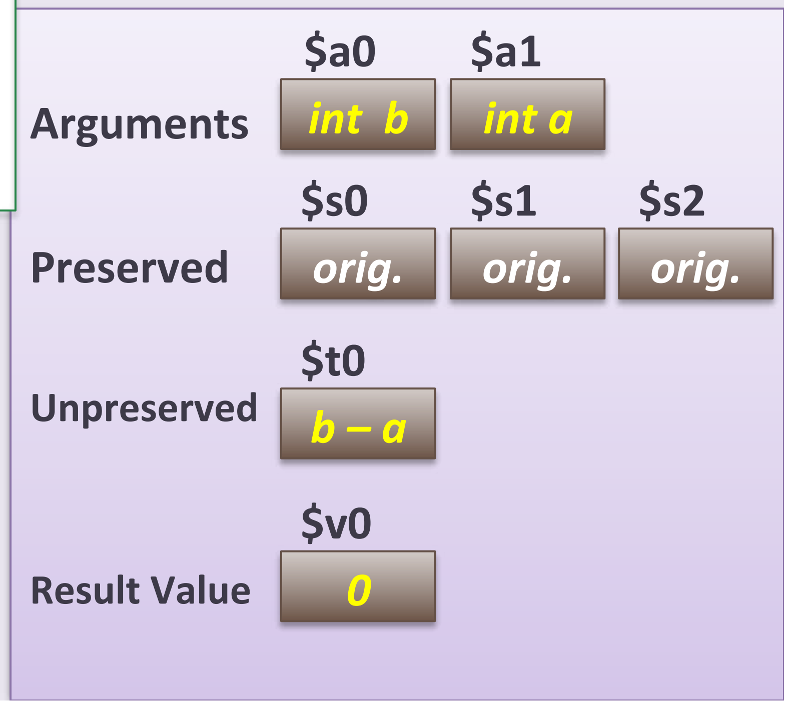
$ra

| | $a0 | $a1 | $s2 |
|---|---|---|---|
| Arguments | *int b* | *int a* | |
| | **$s0** | **$s1** | **$s2** |
| Preserved | *int a* | *int b* | *a – b* |
| | **$t0** | | |
| Unpreserved | *b – a* | | |
| | **$v0** | | |
| Result Value | *b – a* | *0* | |

```
subTwo:
sub $t0, $a0, $a1
move $v0, $t0
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 $ra, 12($sp)
lw $s2, 8($sp)
lw $s1, 4($sp)
lw $s0, 0($sp)
addiu $sp, $sp, 16

jr $ra
```

```c
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;
}
```



# Lessons Learned

- We passed arguments into the functions using **$a***
- We used **$s*** to work out calculations in registers ***that we wanted to preserve***, so we made sure to save them in the call stack
  - These var values DO need to live beyond a call
  - In the end, the original values were returned back
- We used **$t*** to work out calcs. in regs ***that we did not need to preserve***
  - These values DO NOT need to live beyond a function call
- We used **$v*** as regs. to return the value of the function

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

# Another Example Using Recursion

# Recursive Functions

- This same setup handles nested function calls and recursion
  - i.e. By saving `$ra` methodically on the stack

- 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* (the two if tests)

```
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**
  - The output (or partial output) to the call
  - The value of $ra (since this is a recursive function)

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

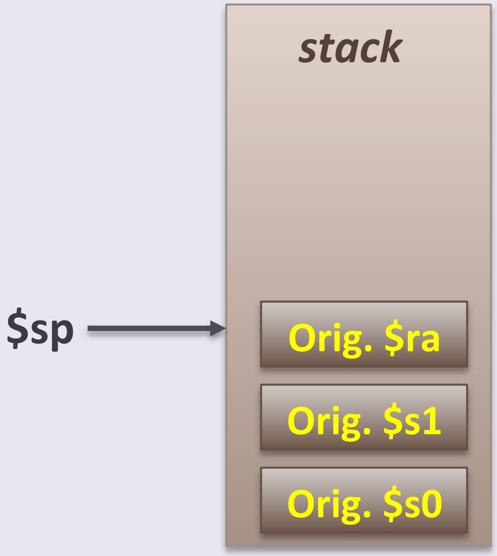
- Then we need to save `$s0`, `$s1` and `$ra` on the stack
  - So that we do not corrupt/lose what's already in these regs
- We'll use `$s*` registers b/c **we need to preserve them beyond the function call**

# recursive_fibonacci.asm

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

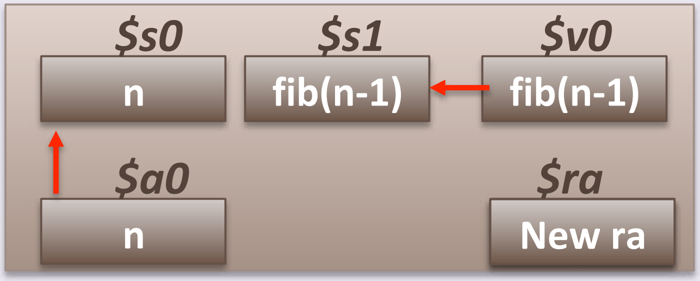

- Next: 3 registers containing integers, means we need to plan for 3 words in the stack
  - **Push** 3 words in (i.e. 12 bytes)
  - **$sp –= 12**
  - The order by which you put them in does *not strictly* matter, but it makes more "organized" sense to

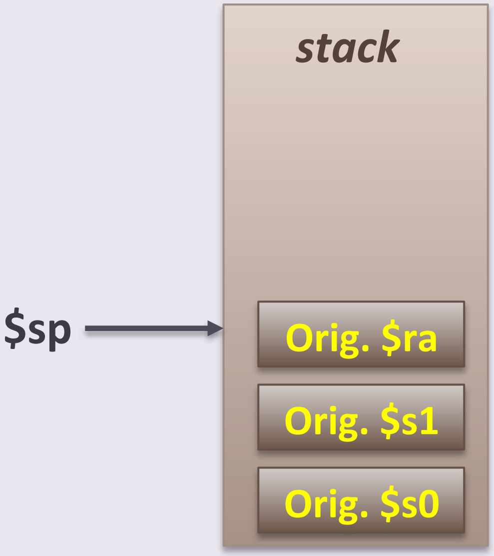
    ***push $s0, $s1, then $ra***



# recursive_fibonacci.asm

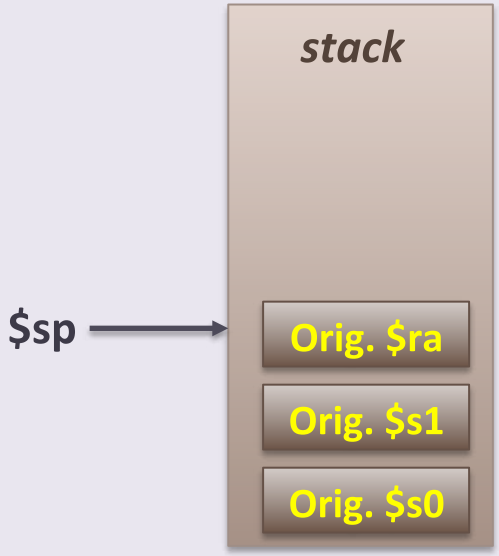
- Next: calculate fib(n – 1)
  - Call recursively, copy output in $s1
- Next: calculate fib(n – 2)



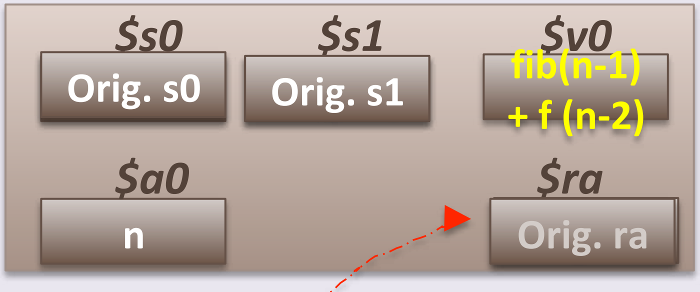
# recursive_fibonacci.asm

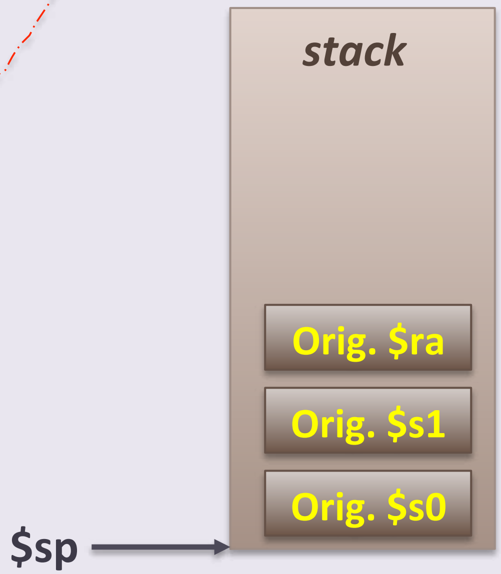
- Next: calculate fib(n – 1)
  - Call recursively, copy output in $s1
- Next: calculate fib(n – 2)
  - Call recursively, add output to $s1

# recursive_fibonacci.asm

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




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

# 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 – 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

- Finish Lab #5 by Friday!
- Take the online mid-term evaluations
  - See upcoming announcement on Piazza
- Next lecture: Digital Logic!

# </LECTURE>