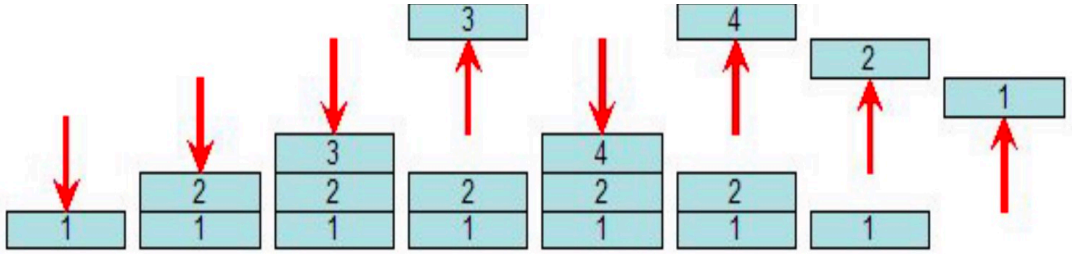


MIPS Calling Convention for Functions, Part 2

CS 64: Computer Organization and Design Logic

Lecture #11

Fall 2019

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB

This Week on "Didja Know Dat?!"

Fun Fact:
Cerf was the inspiration for "The Architect" from The Matrix!

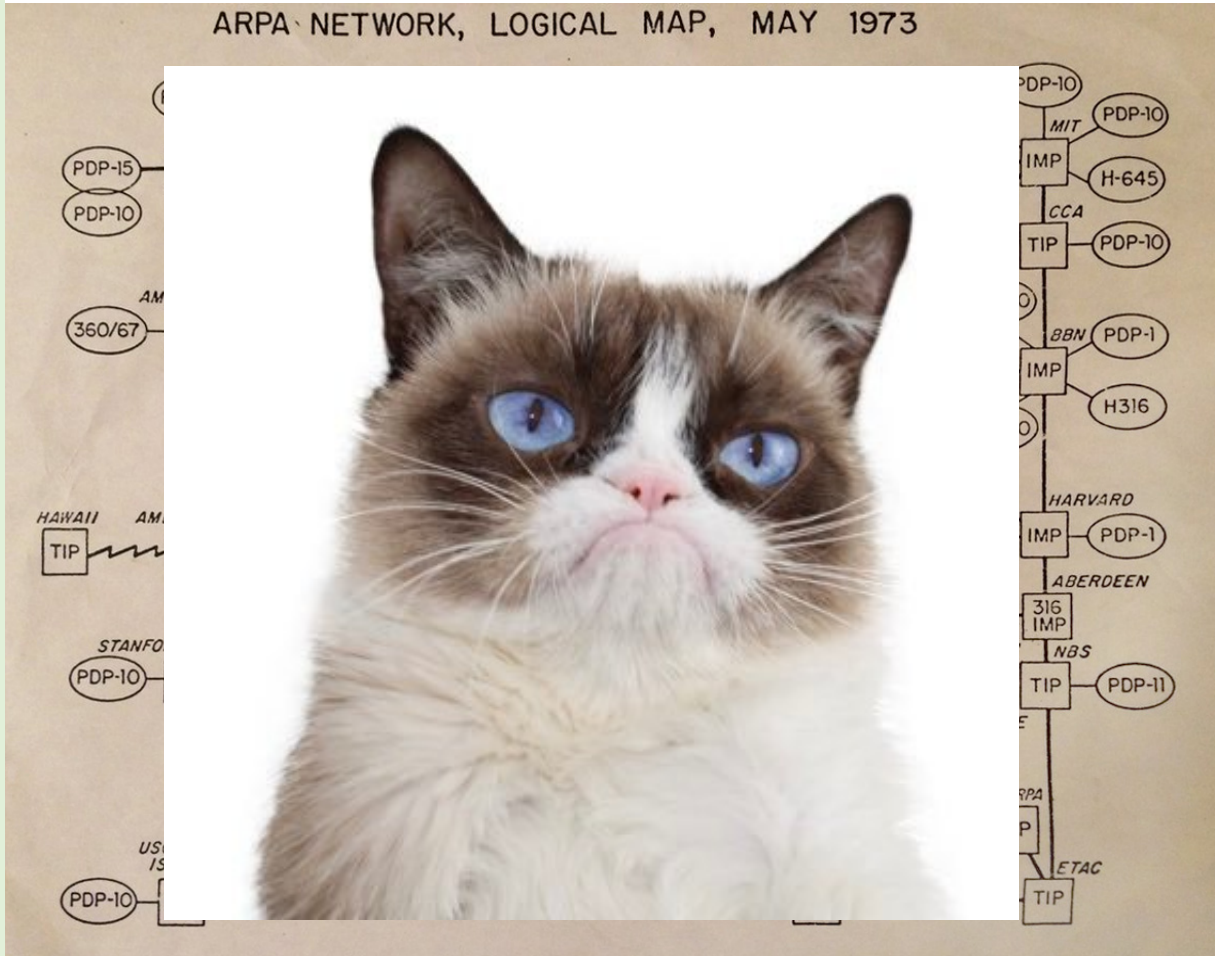
The US Dept. of Defense (DoD) commissioned a research project for a communications system that could sustain operation during partial destruction, such as by nuclear war.

They used their Advanced Research Projects Agency (ARPA) to run this project in the early-mid 1960s. "Packet Switching" was invented.

By the early 1970s, the research project extended across the US, mostly at some universities (including UCSB!!) and was known as *ARPANET*. Thought leaders included *Robert Kahn* and *Vint Cerf* (who invented the TCP/IP protocols).

It evolved in to NSFNET (80s) and then the Internet (90s) as it became more and more part of the public domain.

That's how we can now exchange pictures of cats!



Administrative

- Lab 5 due **FRIDAY**!
- Midterm on Wednesday!

What's on the Midterm?

What's on It?

- Everything we've done so far from start to Monday, 11/4
- *Exception:* no digital logic design

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

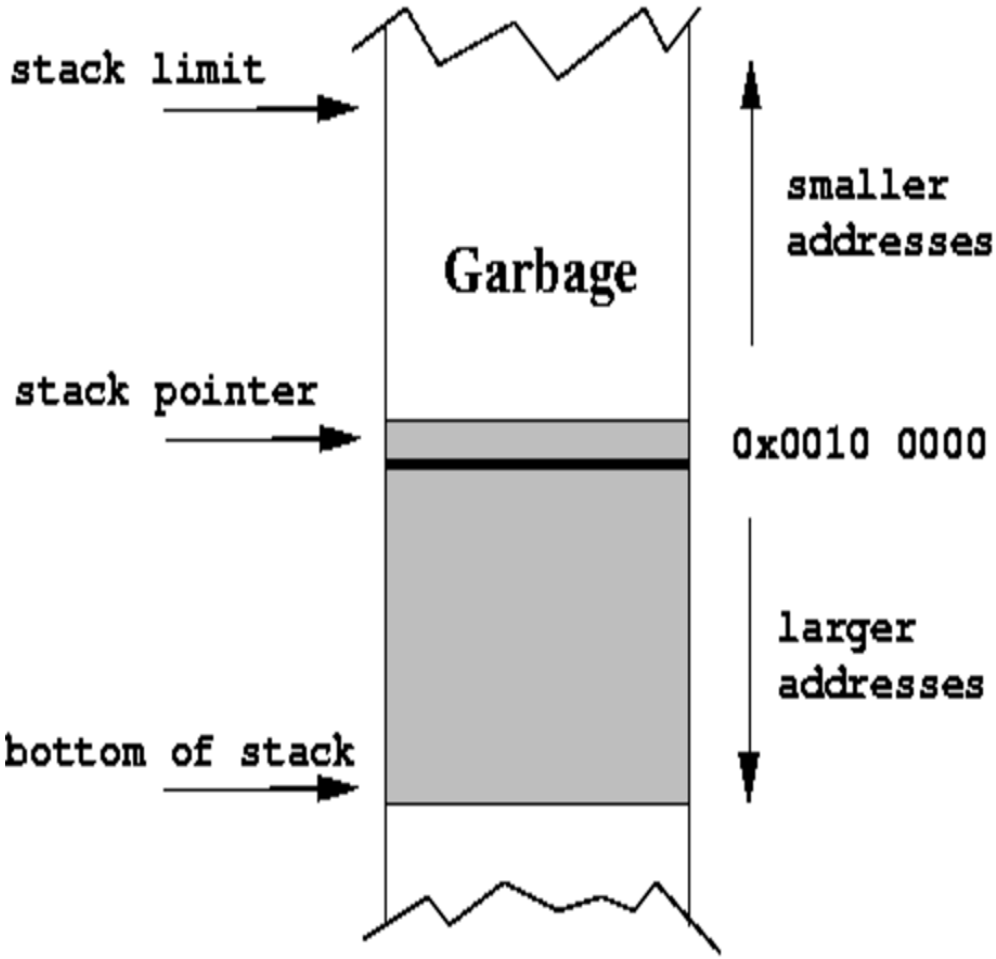
Lecture Outline

- More Examples with MIPS CC

Any Questions From Last Lecture?

Reminder: How the Stack Works

- *Upon reset, $sp points to the "bottom of the stack" – the largest address for the stack*
  - *(0x7FFF FFFC, see MIPS RefCard)*
- *As you move $sp, it goes from high to low address*
- *The "top of the stack" is the stack limit*
  - *(0x1000 8000, see MIPS RefCard)*

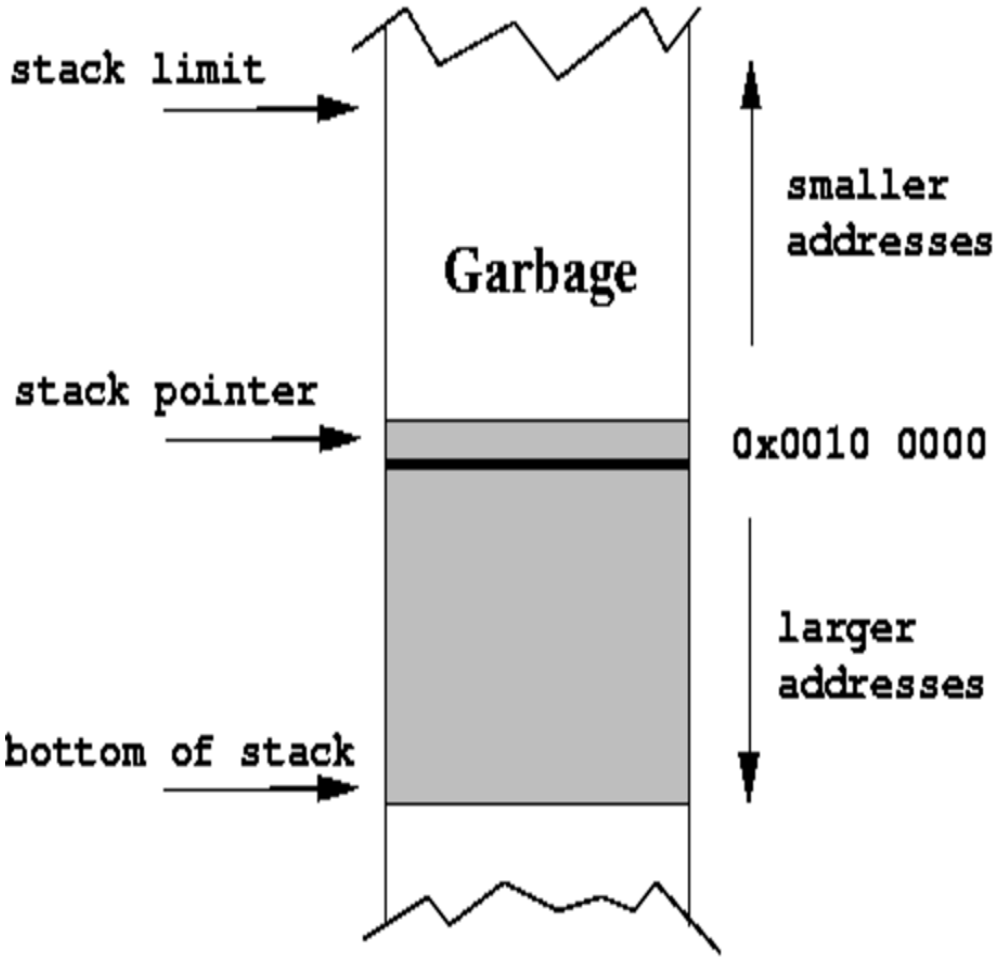


Reminder: How the Stack Works

- *When you want to store some N registers into the stack, the **convention** says you must:*

  A. *Make room in the stack (i.e. move $sp **4xN** places)*

  B. *Then store words accordingly*

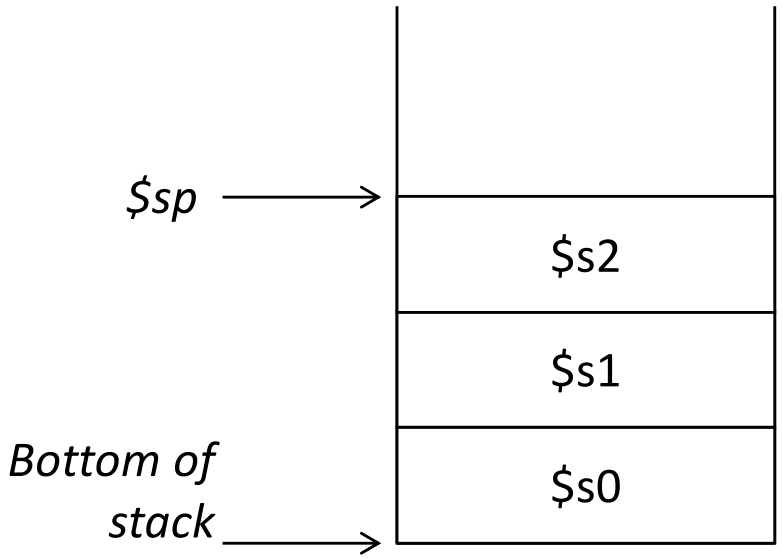


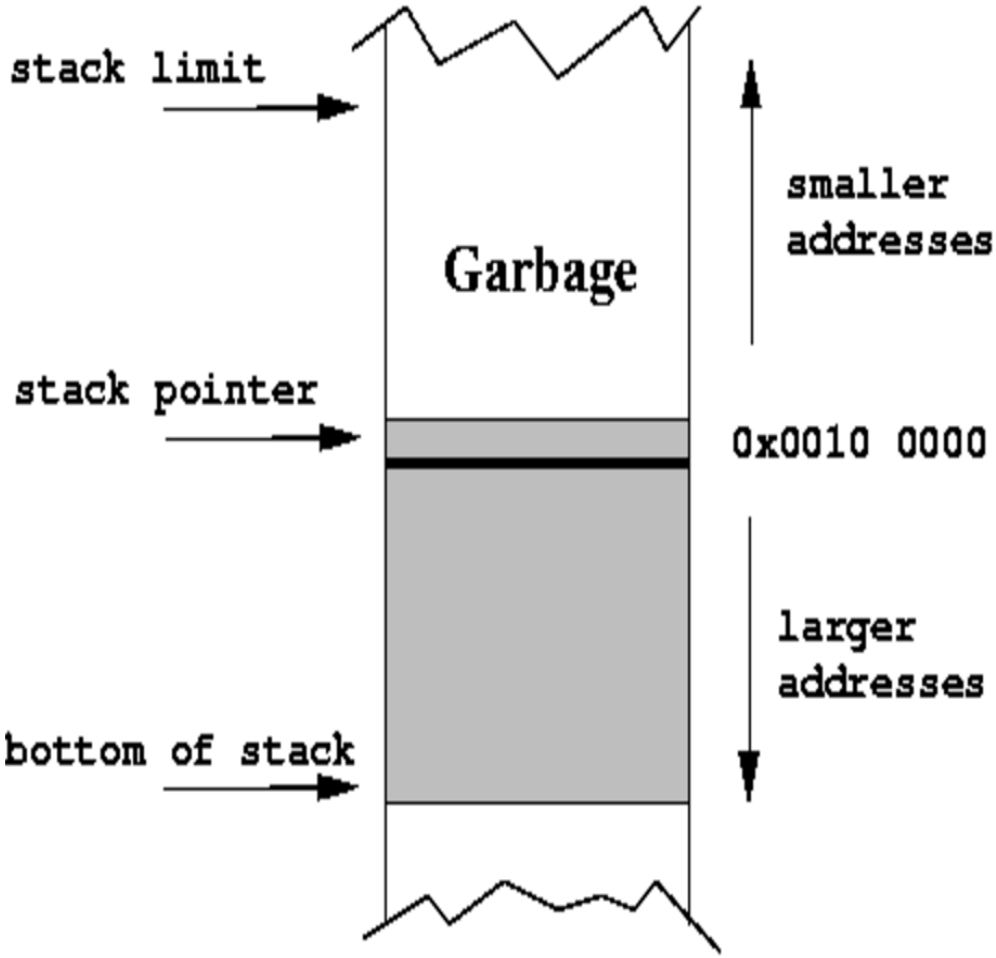
Reminder: How the Stack Works

Example:

You want to store $s0, $s1, and $s2:

```
addiu $sp, $sp, -12      # 'cuz 3 x 4 = 12
sw $s0, 8($sp)
sw $s1, 4($sp)
sw $s2, 0($sp)
```





The MIPS Convention In Its Essence

- Remember: **Preserved** vs **Unpreserved** Regs
- **Preserved**: **$s0 - $s7**, and **$ra**, and **$sp (by default)**
- **Unpreserved**: **$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**?

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

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****.*

```
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
```

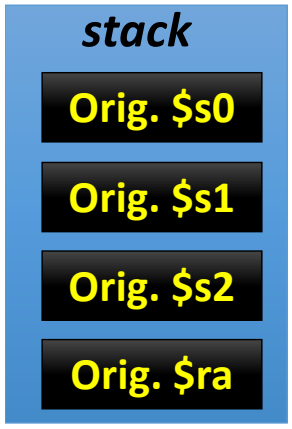
```
move $a0, $s1
move $a1, $s0

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
```

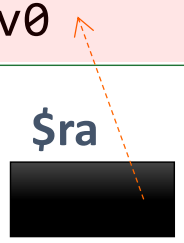
```
int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

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

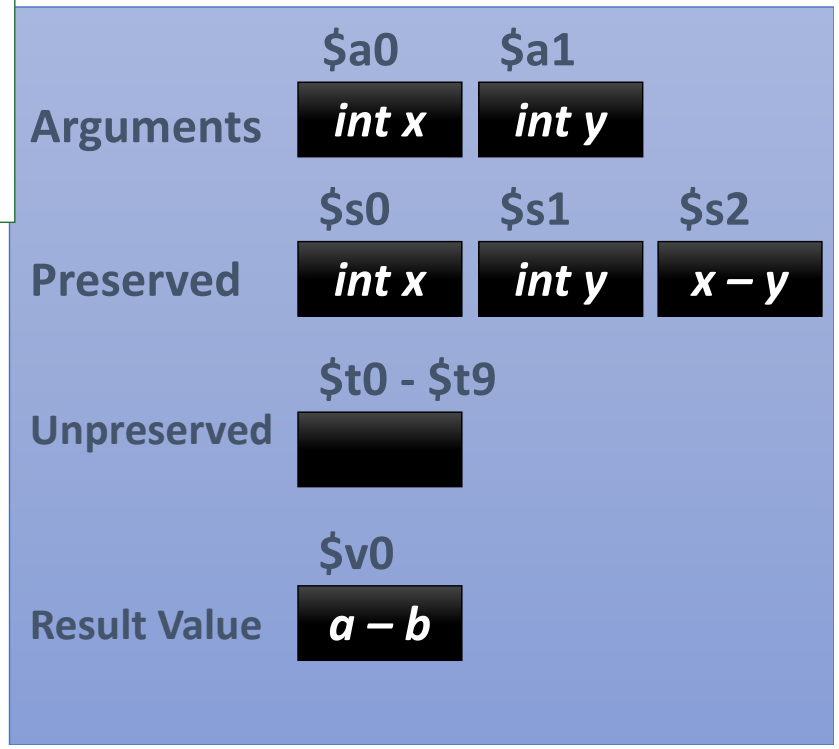
```
subTwo:
sub $v0, $a0, $a1
jr $ra

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;
}
```

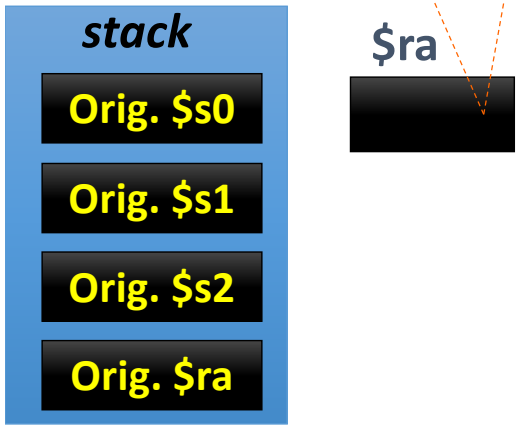


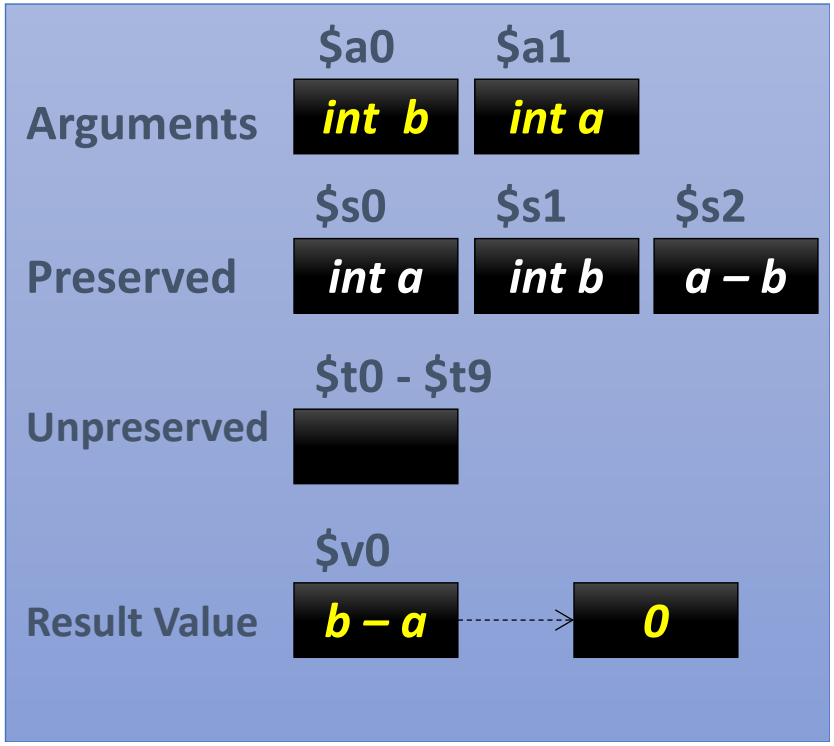




```
subTwo:
sub $v0, $a0, $a1
jr $ra

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;
}
```



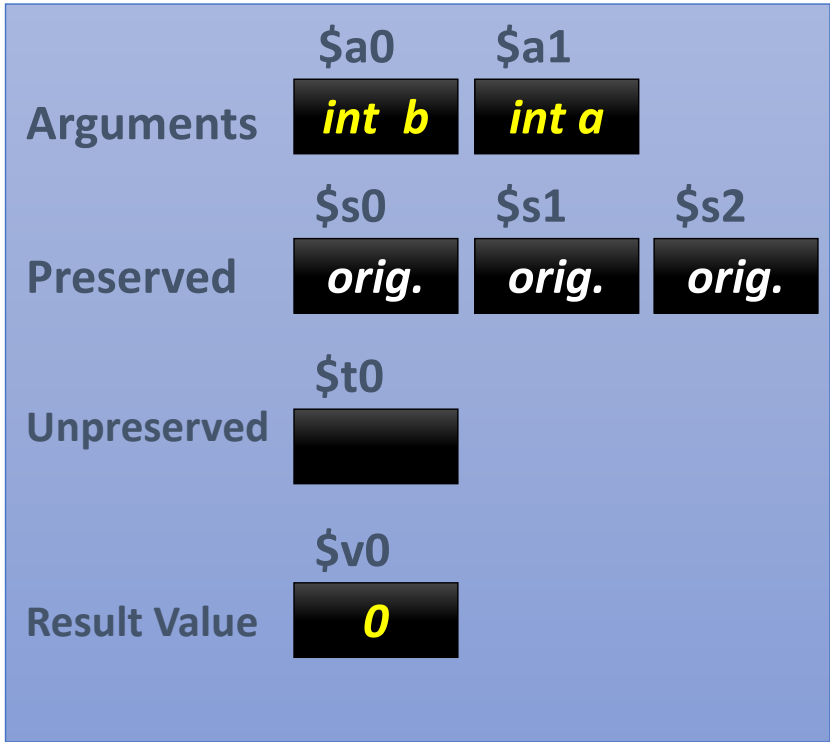


```
subTwo:
sub $v0, $a0, $a1
jr $ra

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;
}
```



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 *could* use **`$t*`** to work out some calcs. in regs ***that we did <u>not</u> need to preserve***
  - These values DO NOT need to live beyond a function call
- We used **`$v*`** as regs. to return the value of the function

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

```
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)

- 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.

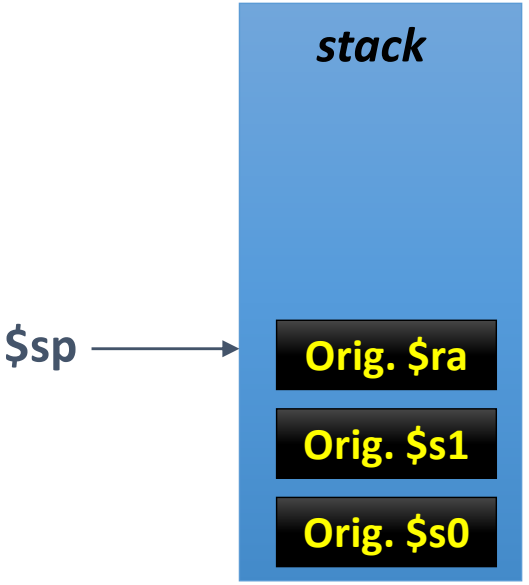
```
li $a0, 7
jal fibonacci
```

recursive_fibonacci.asm
Inside the function "fibonacci"

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

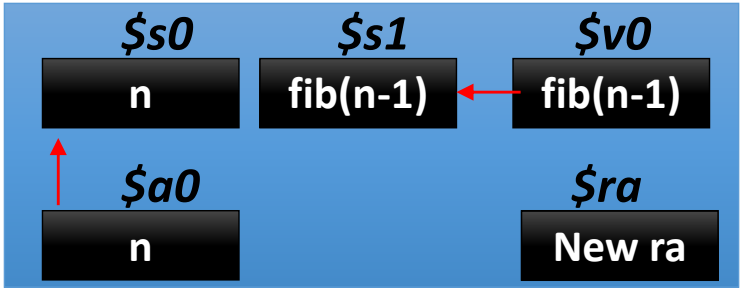
- Next: Do the recursion --- but first…!

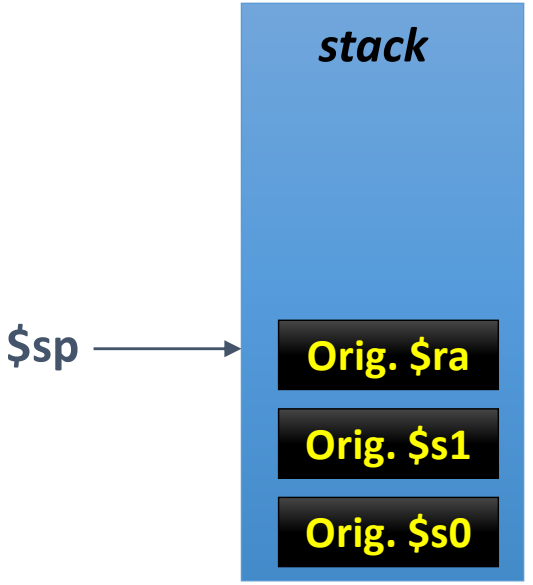
  We need to plan for 3 words in the stack
  - **$sp = $sp – 12**
  - **Push** 3 words in (i.e. 12 bytes)
  - The order by which you put them in does *not strictly* matter, *but* it makes more "organized" sense to ***push $s0, then $s1, then $ra***



recursive_fibonacci.asm

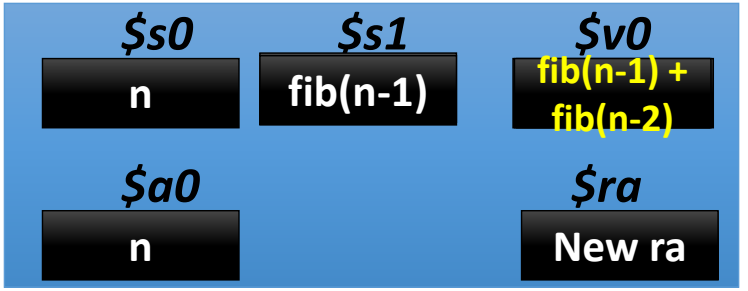
- Next: calculate fib(n – 1)
  - Call recursively & copy output ($v0) in `$s1`
- Next: calculate fib(n – 2)

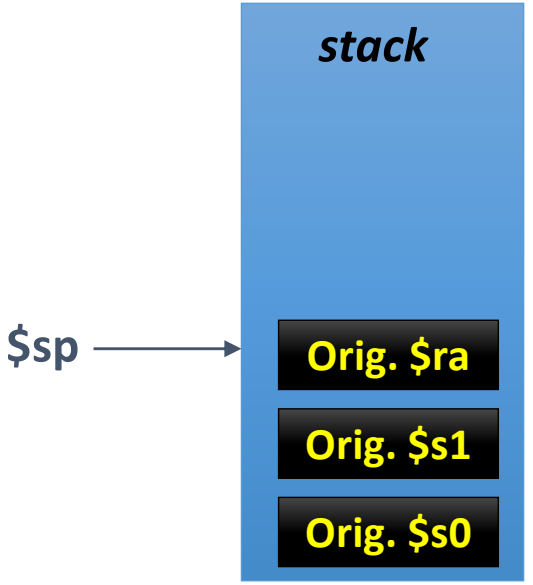




recursive_fibonacci.asm

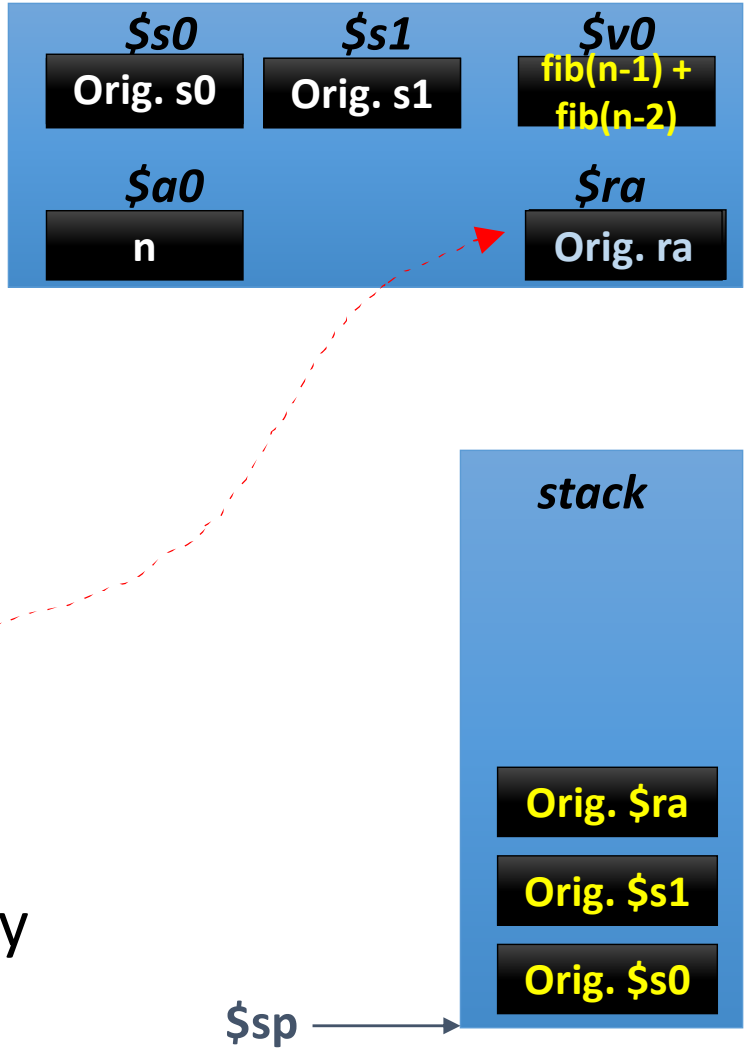
- Next: calculate fib(n – 1)
  - Call recursively & copy output ($v0) in $s1
- Next: calculate fib(n – 2)
  - Call recursively & add $s1 to the output ($v0)





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 **Wednesday**
- *Next time: Intro to Digital Logic*


</LECTURE>