

Function Calling Conventions

CS 64: Computer Organization and Design Logic
Lecture #9

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

- More on MIPS Calling Convention
 - Functions calling functions
 - The role of the convention and how to use it

Any Questions From Last Lecture?

Calling Functions on MIPS

- Two crucial instructions: **jal** and **jr**
- One specialized register: **\$ra**
- **jal (jump-and-link)**
 - Simultaneously **jump to an address**, and **store the location of the next instruction** in register **\$ra**
- **jr (jump-register)**
 - Jump to the address stored in a register, often **\$ra**

MIPS Calling Conventions

- What we are talking about today and the next lecture
- You **must** read the handout on **MIPS Calling Conventions**
 - Found on our webpage

CS64, Spring 2018

Course Information

- [0. Calendar](#)
- [1. Syllabus, CMPSC 64, Spring 2018](#)
- [2. Demo Code used in Lecture](#)
- [3. Class Grades](#)
- [4. Class Readings](#)
- [5. MIPS Reference Card PDF](#)
- [6. MIPS Calling Convention](#)
- [7. Exam Reviews](#)



Passing and Returning Values in MIPS

- Registers **\$a0** thru **\$a3**
 - **Argument registers**, for passing function arguments
- Registers **\$v0** and **\$v1**
 - **Return registers**, for passing return values

Function Calls Within Functions...

Given what we've said so far...

- What about this code makes our previously discussed setup *break*?
 - You would need
multiple copies of \$ra
- You'd have to copy the value of \$ra to *another* register (or to mem) before calling another function
- Danger: You could run out of registers!

```
void foo() {  
    bar();  
}  
void bar() {  
    baz();  
}  
void baz() {}
```

Another Example...

What about this code makes this setup break?

- Can't fit all variables in registers at the same time!
- How do I know which registers are even usable without looking at the code?

```
void foo() {  
    int a0, a1, ..., a20;  
    bar();  
}  
void bar() {  
    int a21, a22, ..., a40;  
}
```


Solution??!!

- Store certain information in memory only at certain times
- Ultimately, this is where the **call stack** comes from
- So what (registers/memory) save what???

What Saves What?

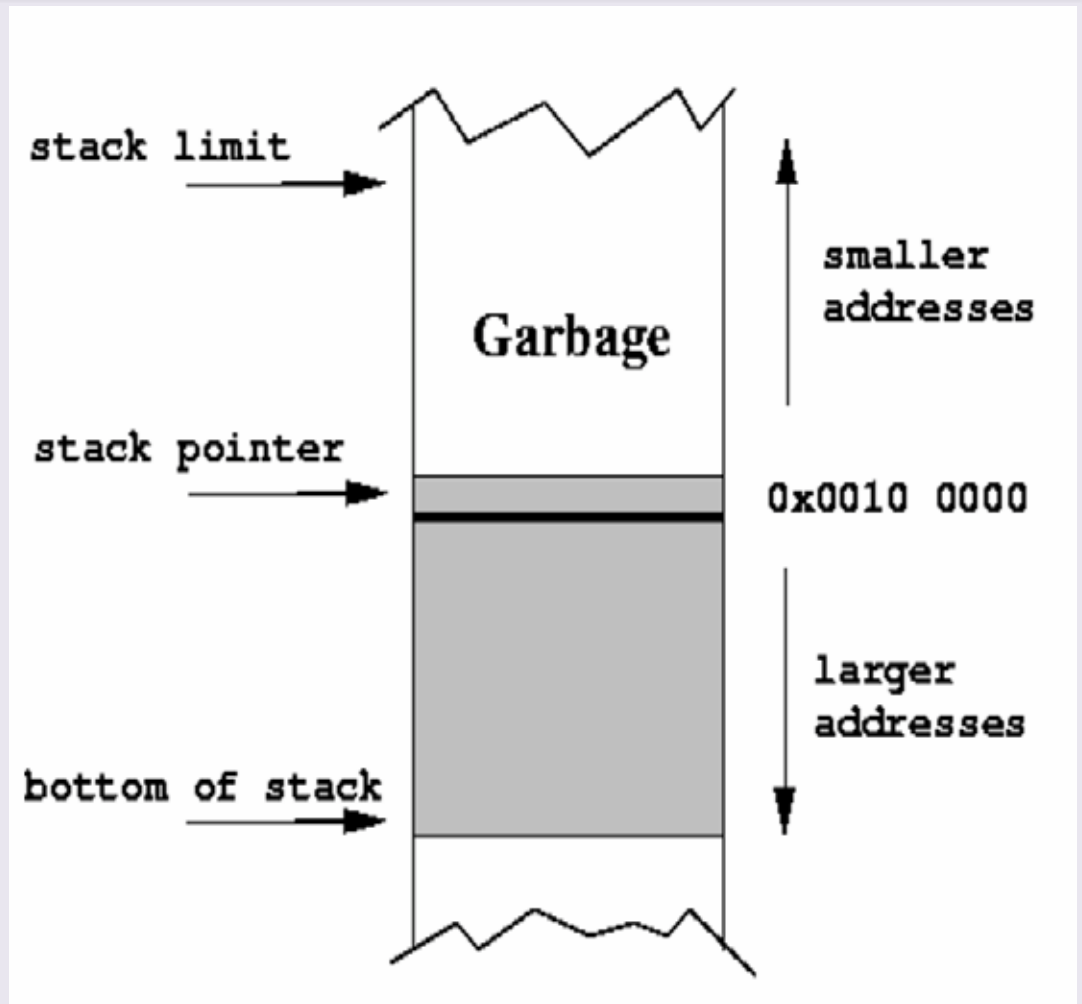
- By MIPS convention, certain registers are *designated* to be preserved across a call
- Preserved registers are saved by the ***function called*** (e.g., \$s0 - \$s7)
 - So these are saved at the start of every function
- Non-preserved registers are saved by the ***caller of the function*** (e.g., \$t0 - \$t9)
 - So these are saved by the function's caller
 - Or not... (they can be ignored under certain circumstances)

And Where is it Saved?

- Register values are saved on the **stack**
- The top of the stack is held in **\$sp** (**stackpointer**)
- The stack grows
from high addresses to low addresses

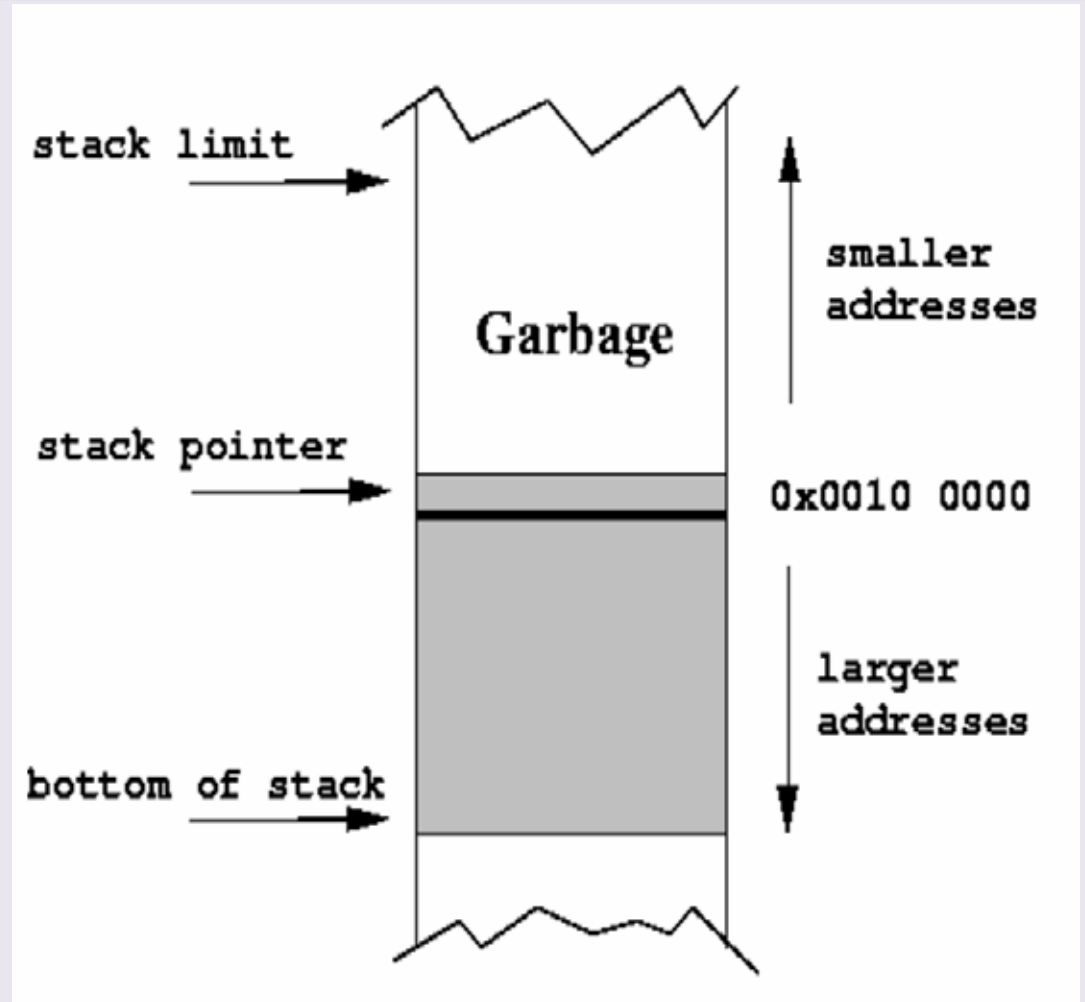
The Stack

When a program starts executing, a certain *contiguous* section of memory is set aside for the program called the stack.



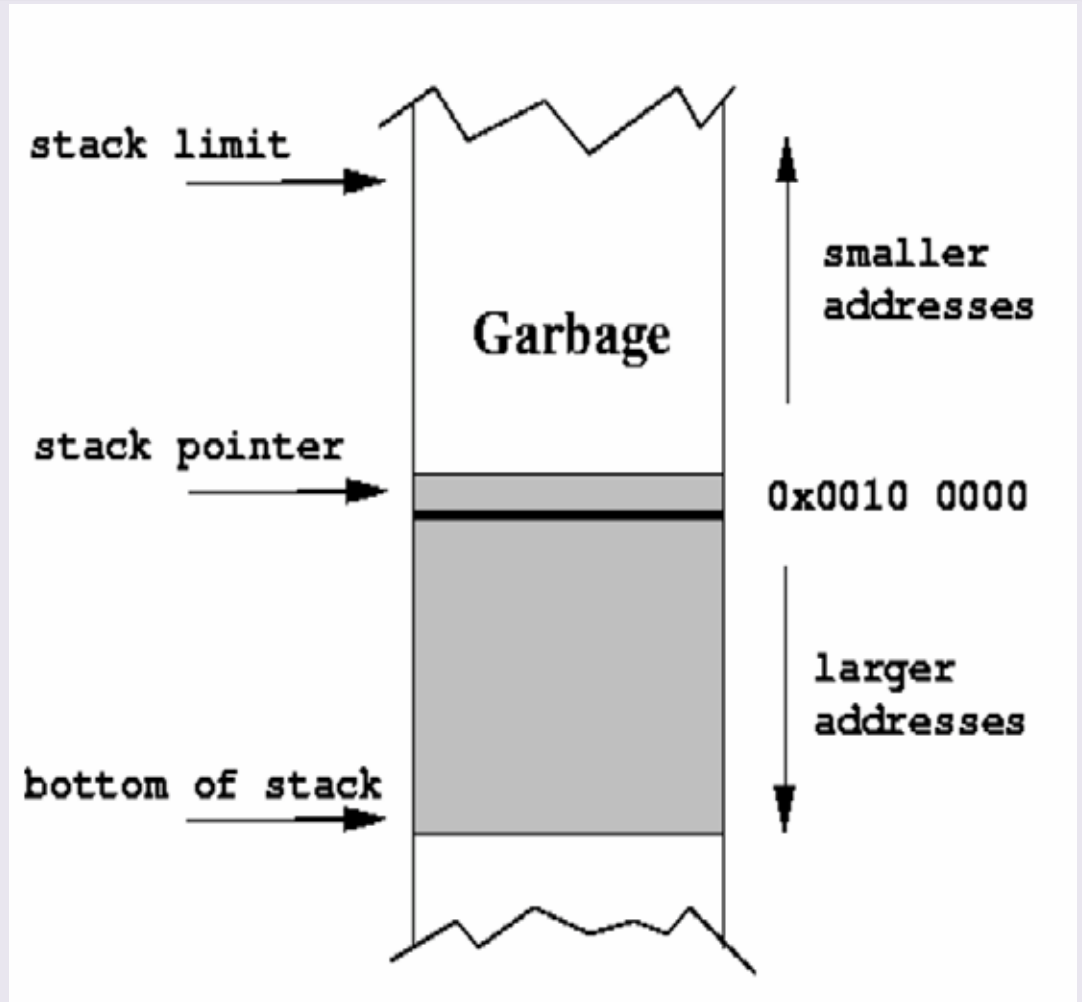
The Stack

- The **stack pointer** is a register ($\$sp$) that contains the **top of the stack**.
- $\$sp$ contains the *smallest address x* such that any address smaller than x is considered **garbage**, and any address greater than or equal to x is considered **valid**.



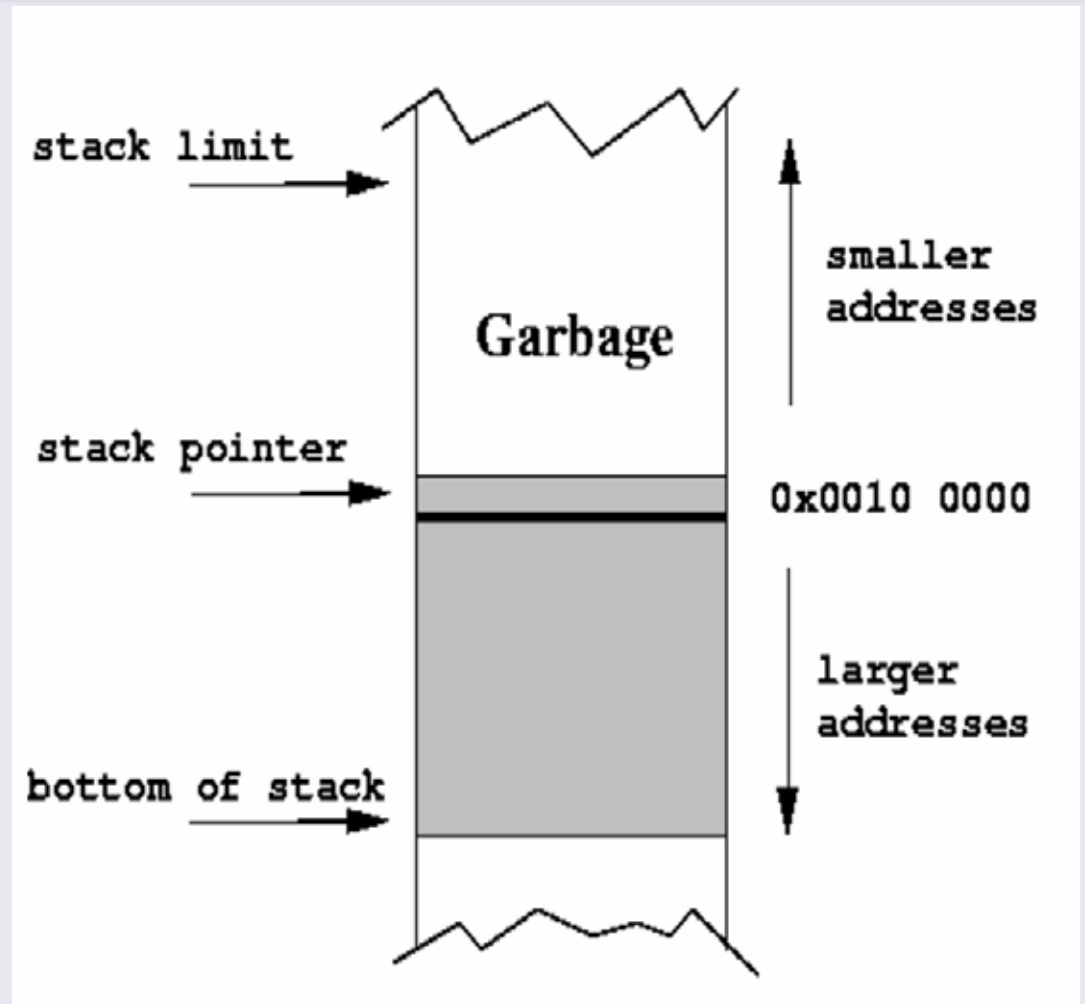
The Stack

- In this example, **\$sp** contains the value **0x0000 1000**.
- The shaded region of the diagram represents **valid** parts of the stack.

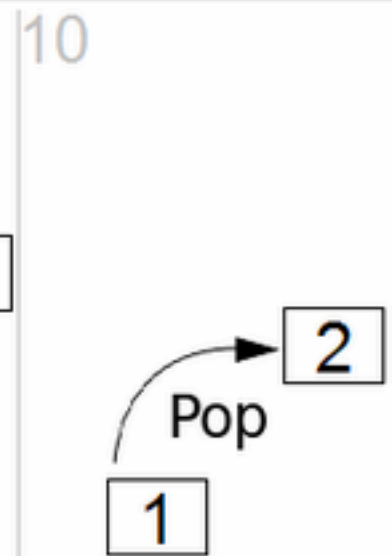
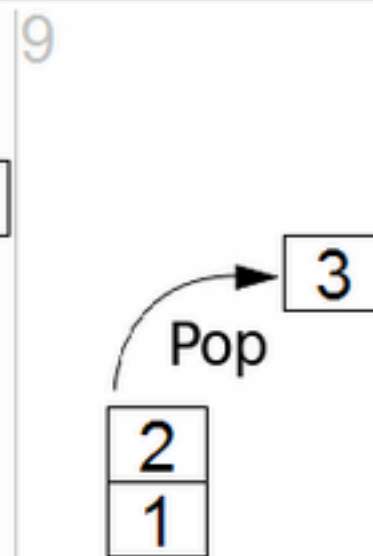
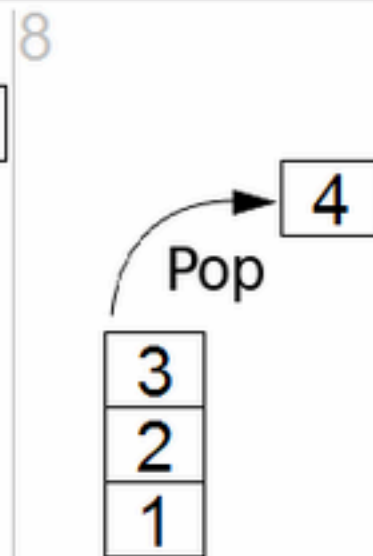
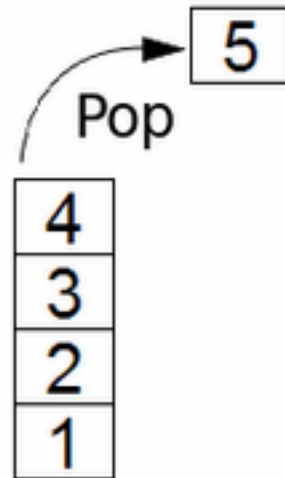
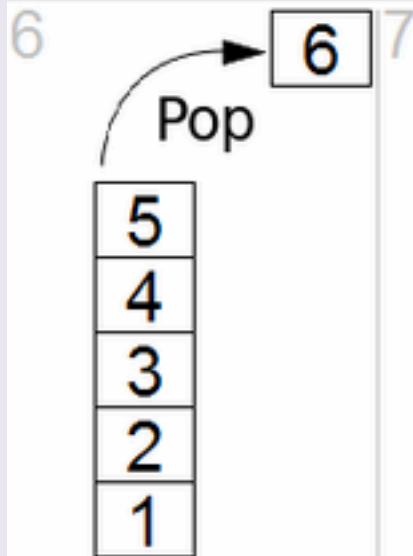
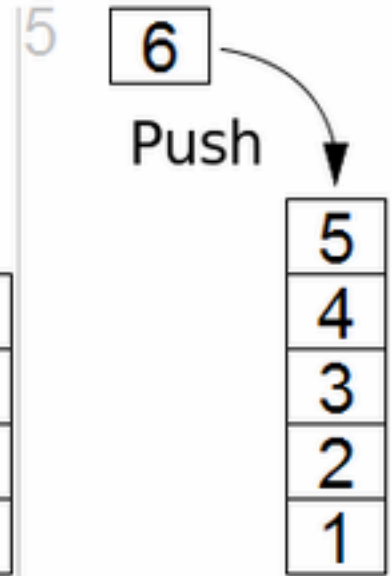
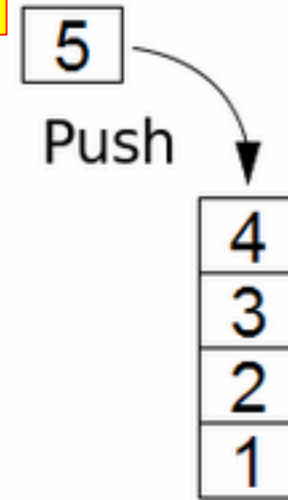
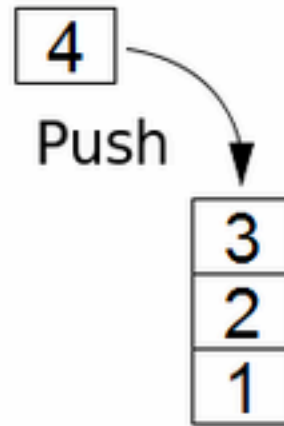
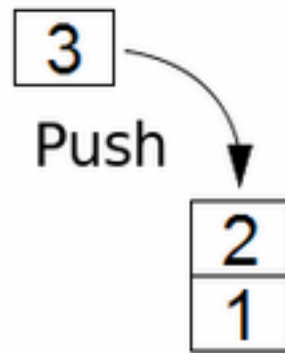
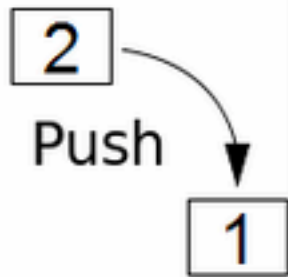


The Stack

- **Stack Bottom:** The largest valid address of a stack.
- When a stack is initialized, `$sp` points to the stack bottom.
- **Stack Limit:** The smallest valid address of a stack.
- If `$sp` gets smaller than this, then there's a **stack overflow**



STACK (LIFO) PUSH AND POP



Stack Push and Pop

- **PUSH** one or more registers
 - Subtract 4 times the number of registers to be pushed on the stack pointer
 - *Why????*
 - Copy the registers *to* the stack (do a **sw** instruction)
- Example:

```
addi $sp, $sp, -8 # 2 registers to save
sw $s0, 4($sp)
sw $s1, 0($sp)
```



Stack Push and Pop

- **POP** one or more registers
 - Reverse process from **push**
 - Copy the data *from* the stack to the registers (do a **lw** instruction)
 - Add 4 times the number of registers to be popped on the stack.

Example:

```
lw $s0, 4($sp)
```

```
lw $s1, 0($sp)
```

```
addi $sp, $sp, 8    # 2 registers to restore  
# you can do the addi first - it's ok
```



save_registers.asm

- The program will look at 2 integers (a0, a1) and ultimately returns $(a0 + a0) + (a1 + a1)$ via a function call (i.e. **jal**)
- The function will first create room for **2 words** on the stack
 - It will **push \$s0 & \$s1** onto the stack
 - We'll use **\$s0** and **\$s1**
b/c we want them to be preserved across a call
- It will calculate the returned value and put the result in **\$v0**
- We will then restore the original registers
 - It will **pop** 2 words from the stack & place them in **\$s0 & \$s1**

```
.data
solution_text: .asciiz "Solution: "
saved_text:    .asciiz "Saved: "
newline:      .asciiz "\n"
.text
# $a0: first integer
# $a1: second integer
# Returns ($a0 + $a0) + ($a1 + $a1) in $v0.
# Uses $s0 and $s1 as part of this process because these are preserved across a call.
# add_ints must therefore save their values internally using the stack.
add_ints:
    # save $s0 and $s1 on the stack (i.e. push)
    addi $sp, $sp, -8 # make room for two words
    sw $s0, 4($sp)    # note the non-zero offset
    sw $s1, 0($sp)

# calculate the value
    add $s0, $a0, $a0
    add $s1, $a1, $a1
    add $v0, $s0, $s1

# because $t0 is assumed to not be preserved, we can modify it directly (and it will not
# matter b/c we'll pop the saved $t0 out of the stack later)
    li $t0, 4242

# restore the registers and return (i.e. pop)
    lw $s1, 0($sp)
    lw $s0, 4($sp)
    addi $sp, $sp, 8
    jr $ra
```

main:

```

# We “happen” to have the value 1 in $t0 and 2 in $s0 in this example
# $t0 and $s0 are independent of the function...
li $t0, 1
li $s0, 2
# We want to call add_ints. Because we want to save the value of $t0, in this case,
# and because it's not preserved across a call (we can't assume it will be), it is our
# (the caller's) responsibility to store it on the stack and restore it afterwards
addi $sp, $sp, -4
sw $t0, 0($sp)  # saving $t0 is the caller's responsibility, $s0 is the callee's...

# setup the function call and make it
li $a0, 3
li $a1, 7
jal add_ints

# restore $t0 - also, we can “assume” that $s0 still has the value 2 in it
# because the CC says the function has to preserve $s registers
lw $t0, 0($sp)
addi $sp, $sp, 4

# print out the solution prompt
move $t1, $v0
li $v0, 4
la $a0, solution_text
syscall

# print out the solution itself
li $v0, 1
move $a0, $t1
syscall

# print out a newline and end (not shown)
la $a0, newline
li $v0, 4
syscall

```

What is a Calling Convention?

- It's a **protocol** about *how* you call functions and *how* you are supposed to return from them
- Every CPU architecture has one
 - They can differ from one arch. to another
- 3 Reasons why ***we*** care:
 - Because it makes programming a lot easier if everyone agrees to the same consistent (i.e. reliable) methods
 - Makes **testing** a whole lot easier
 - I will ask you to use it in assignments and in exams!
 - And you loose major points (or all of them) if you don't...

More on the “Why”

- Have a way of implementing functions in assembly
 - But not a clear, easy-to-use way to do complex functions
- In MIPS, we do not have an *inherent* way of doing **nested/recursive functions**
 - Example: Saving an *arbitrary amount* of variables
 - Example: Jumping back to a place in code *recursively*
- There is more than one way to do things
 - But we often need a convention to set **working parameters**
 - Helps facilitate things like testing and inter-compatibility
 - This is partly why MIPS has different registers for different uses

Instructions to Watch Out For

- **jal** <label> and **jar** \$ra always go together
- Function arguments have to be stored ONLY in **\$a0 thru \$a3**
- Function return values have to be stored ONLY in **\$v0 and \$v1**
- If functions need additional registers *whose values we don't care about keeping after the call*, then they can use **\$t0 thru \$t9**
- What about **\$s** registers? AKA the ***preserved registers***
 - Hang in there... will talk about them in a few slides...

MIPS C.C. for CS64: Assumptions

- We will not utilize **\$fp** and **\$gp** regs
 - \$fp: frame pointer
 - \$gp: global pointer
- Assume that functions will not take more than **4** arguments and will not return more than **2** arguments
 - Makes our lives a little simpler...
- Assume that all values on the stack are always 32-bits
 - That is, no overly long data types or complex data structures like C-Structs, Classes, etc...

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; }

```

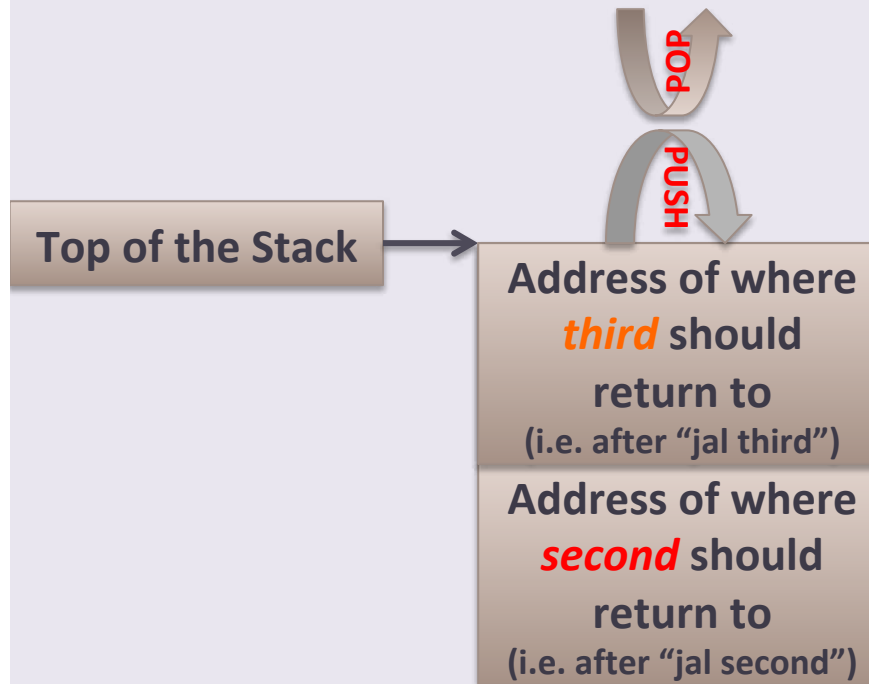
```
void third()
{
    fourth ();
    return; }

```

```
void forth()
{
    return; }

```

MIPS Call Stack



```
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
syscall

```

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

Functions that Call Functions That Need Additional Registers

- Consider this program:
- Can we use **\$t0** and **\$t1** for vars **a** and **b** in **doSomething**?
 - Yes, that's ok...
- Can we use **\$t2** for var. **sub** in **subTwo**?

---NO!--- (why?)

*Because we're supposed to use **\$v0**
for returned values!*

```
int subTwo(int a, int b)
{
    int sub = 2*a - b;
    return sub;
}
```

```
int doSomething(int x, int y)
{
    int a = subTwo(x, y);
    int b = subTwo(3*y, x);
    ...
    return a + b;
}
```

- So, what do we do *if* we have **\$t** registers that now need to be used from one function to another??
 - Not a good place to find yourself, but what if...?

Solution?

Preserve Registers in The Call Stack!

- What did we do in our previous example? `save_registers.asm`
- Values in **unpreserved registers** (every register introduced so far except for `$sp` and `$s` registers), **are not preserved across calls.**
- You must always assume that unpreserved registers **can/will change values across calls**, even if you *know* that they don't..
- Why?
 - Convention. Rules are Rules...
 - **Even if you have correct code**, not obeying these rules means that you don't have fully-independent functions, which leads to **hacky code that may be hard to test** (which is a **cardinal sin in the eyes of the programming world!**)

YOUR TO-DOs

- Make sure you read the handout on **MIPS Calling Conventions**
- Finish assignment/Lab #4
 - Assignment due on FRIDAY
- Prep for Lab #5 on Monday

</LECTURE>