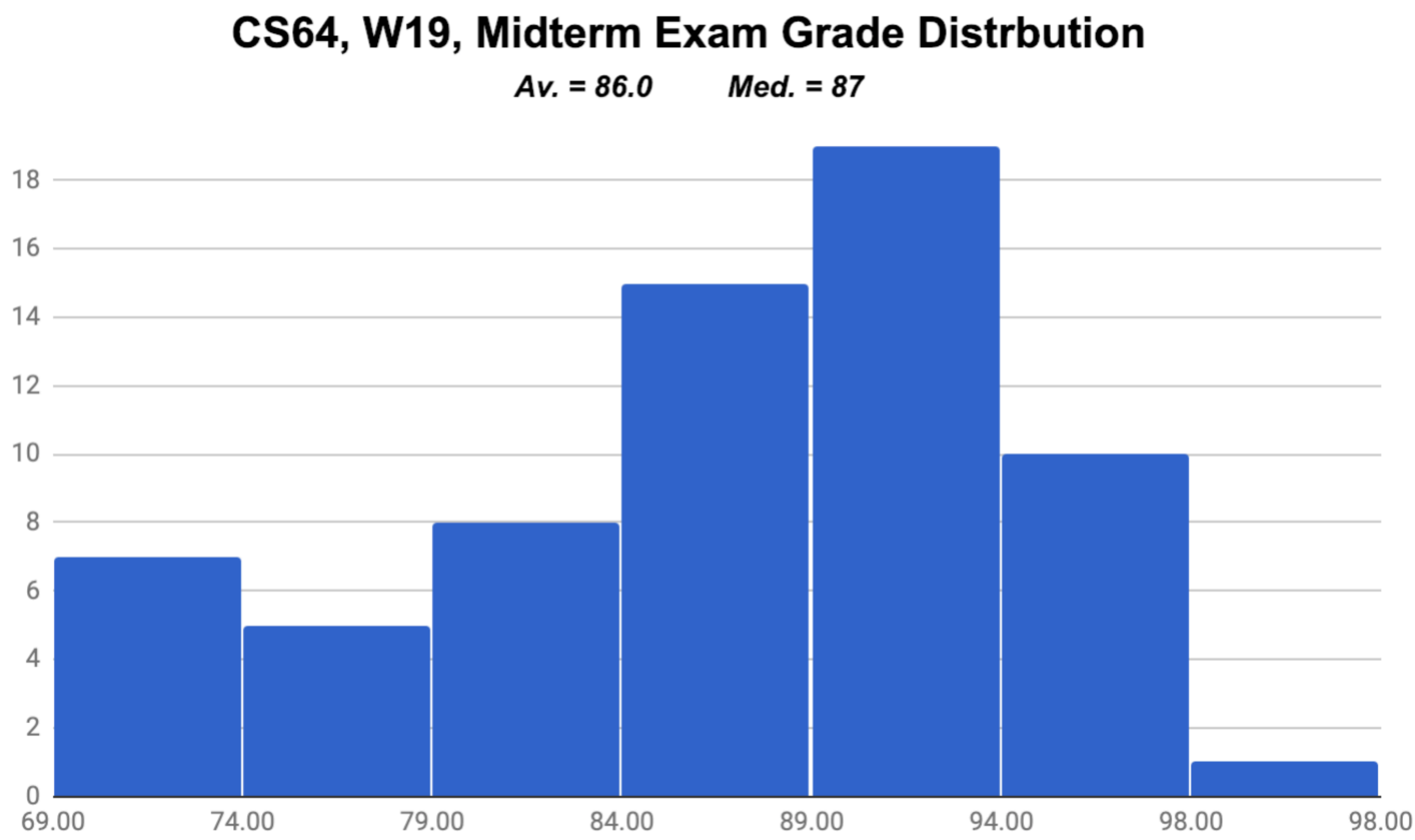
Using the Stack with Functions
Intro to the MIPS Calling Convention

CS 64: Computer Organization and Design Logic
Lecture #9
Winter 2019

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

Administrative

- No lab assignment for this week
- Issue: getting feedback on your assignments
- Midterm has been graded!
  - Grades will be posted on GauchoSpace very soon
  - You can see your exams with your TAs and myself starting next week
    - Will announce on Piazza


CS64, W19, Midterm Exam Grade Distrbution
Av. = 86.0
Med. = 87
18
16
14
12
10
8
6
4
2
0
69.00
74.00
79.00
84.00
89.00
94.00
98.00
98.00

Commonly Seen Mistakes in Midterm

- **Carry Out** versus **Overflow**
- The use of masking and bit-shifting
  - Example: **oppositeOfBitN()** question
- The use of **lw/sw** and the difference between getting a ***value*** from memory and getting an ***address*** from memory
- Programming style: unnecessary code
  - Examples: creating a bunch of zero-valued regs, putting jumps right before the default next line, forgetting jumps

Lecture Outline

- Intro to the MIPS Calling Convention
- Using the stack in MIPS Assembly

Any Questions From Last Lecture?

RECALL: Simple Call Example

- See program: **simple_call.asm**

```
# Calls a function (test) which immediately returns
.text
test:  # return to whoever made the call
       jr $ra

main:  # do stuff…
       # then call the test function
       jal test

exit:  # exit
       li $v0, 10
       syscall
```

*Note: SPIM always starts execution at the line labeled "**main**"*

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**

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

- 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!

Another Example…

What about this code makes this setup break?

- Can't fit all variables in registers at the same time!

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

- How do I know which registers are even usable without looking at the code?

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 should be saved at the start of every function

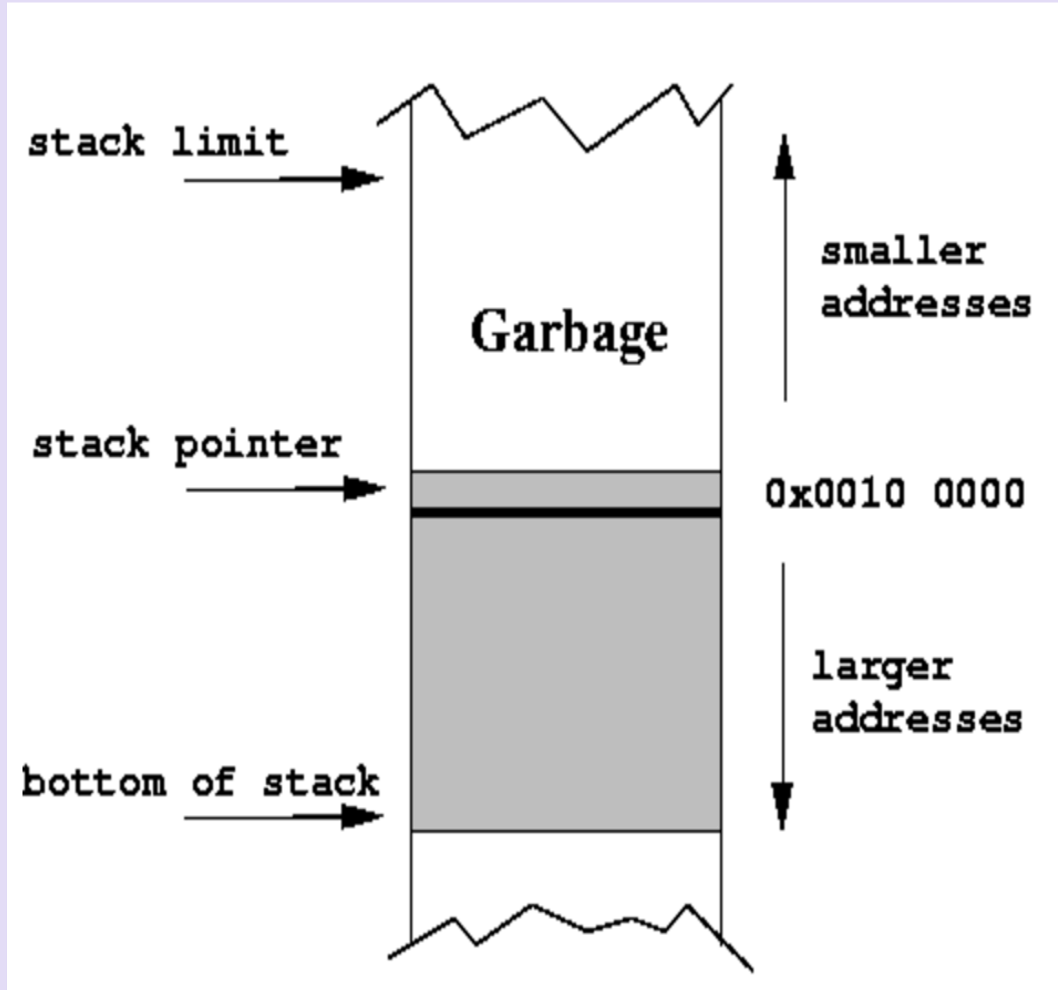
- Non-preserved registers are saved by the ***caller of the function*** (e.g., $t0 - $t9)
  - So these should be saved by the function's caller
  - Or not… (they can be ignored under certain circumstances)

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

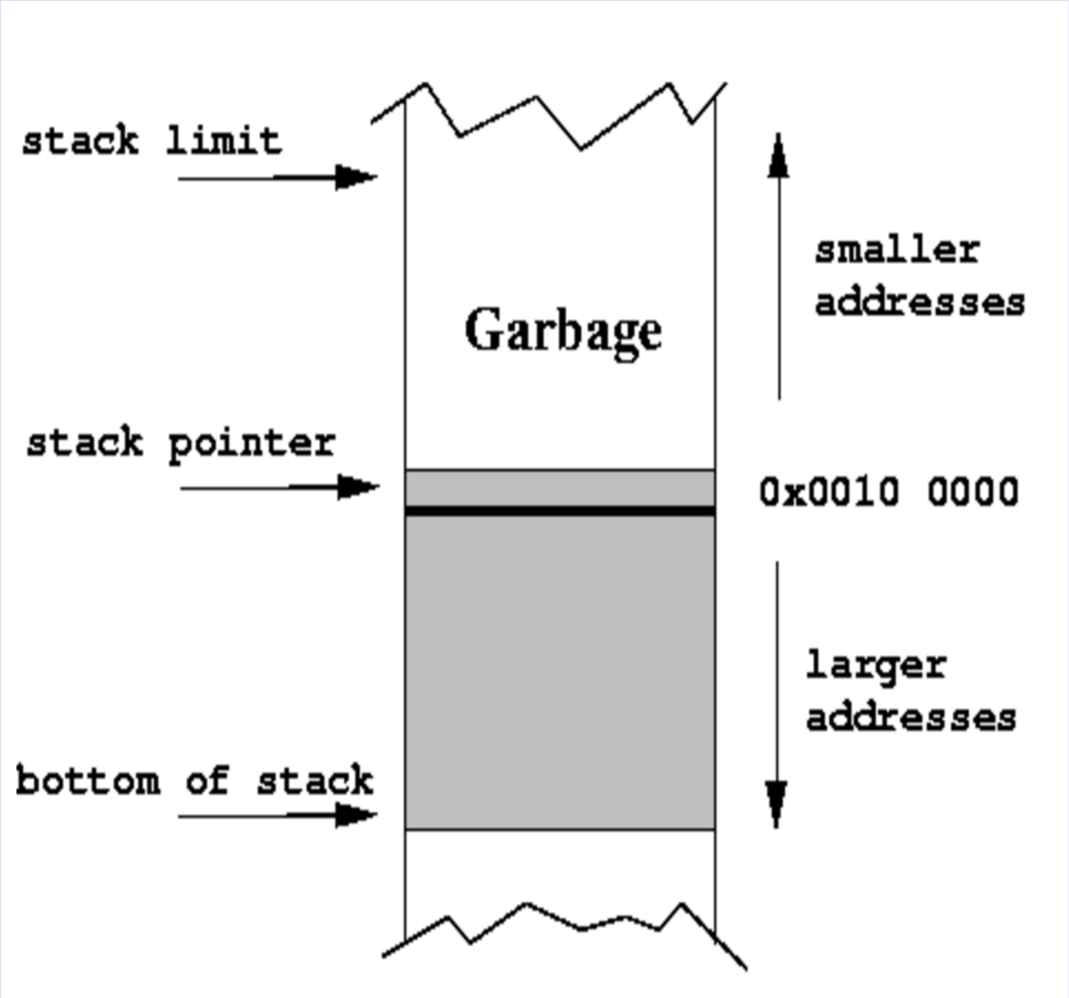
The Stack

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



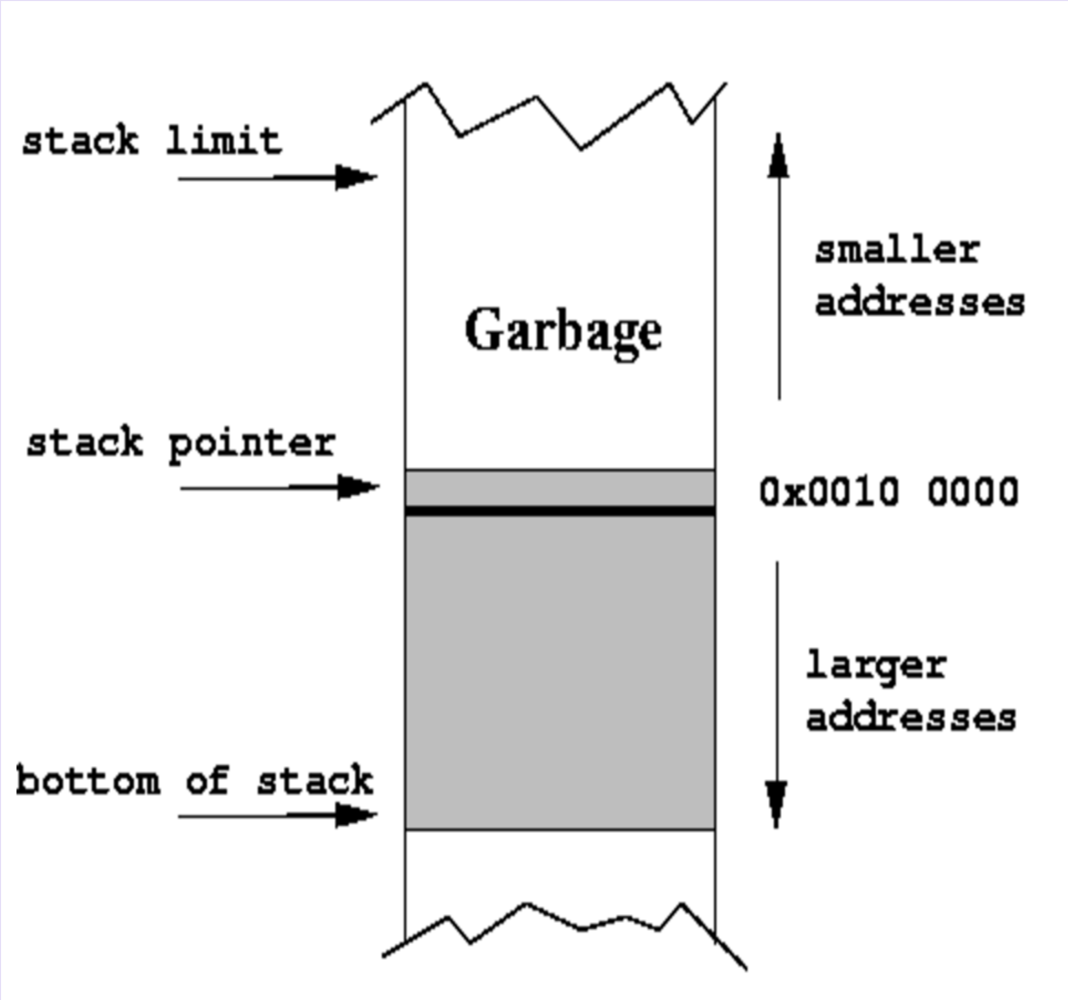
The Stack

- 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***.



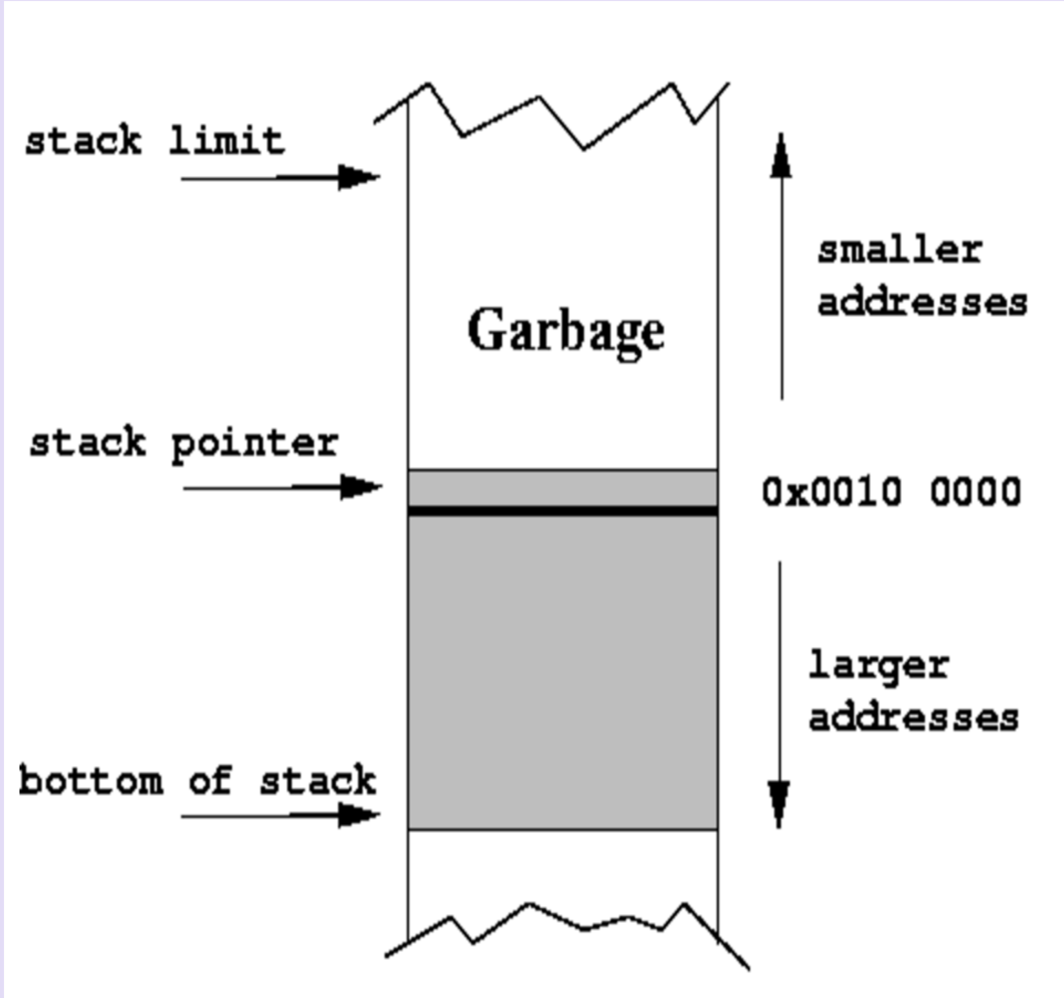
The Stack

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

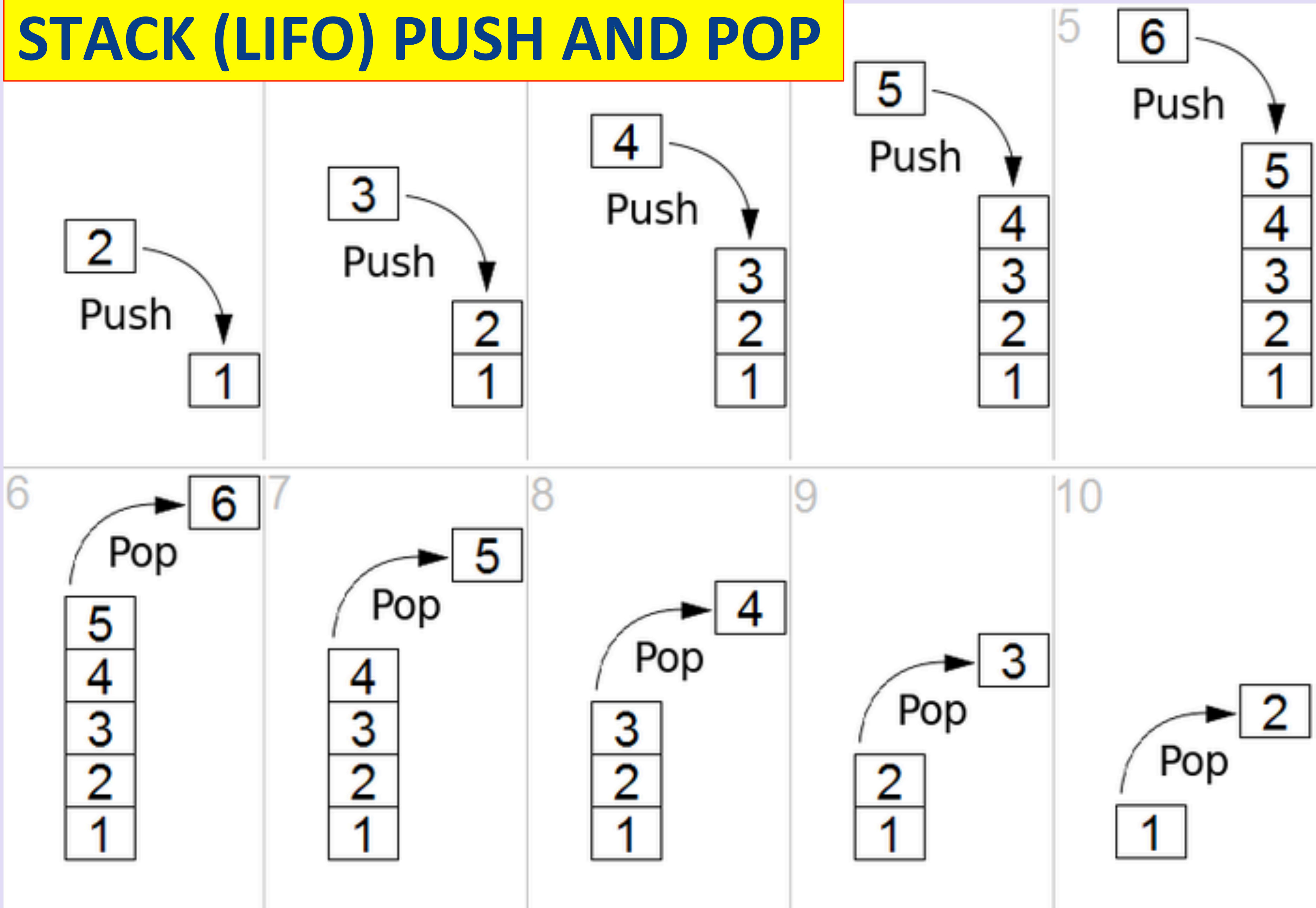


The Stack

- **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 we get a **stack overflow error**



STACK (LIFO) PUSH AND POP



Stack Push and Pop

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

  <u>Example</u>:

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

Stack Push and Pop

- To **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
# Note: you cannot do the addi first
```

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`**

save_registers.asm

```
.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
```

save_registers.asm

```
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 <u>call</u> functions and *how* you are supposed to <u>return</u> 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 <u>complex</u> 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 <u>*is*</u> more than one way to do things
  - But we often need a <u>***convention***</u> 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 **`jr $ra`** always go together
- Function *arguments* have to be stored ONLY in
  `$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***
  - We must save them… more on that…

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…

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

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

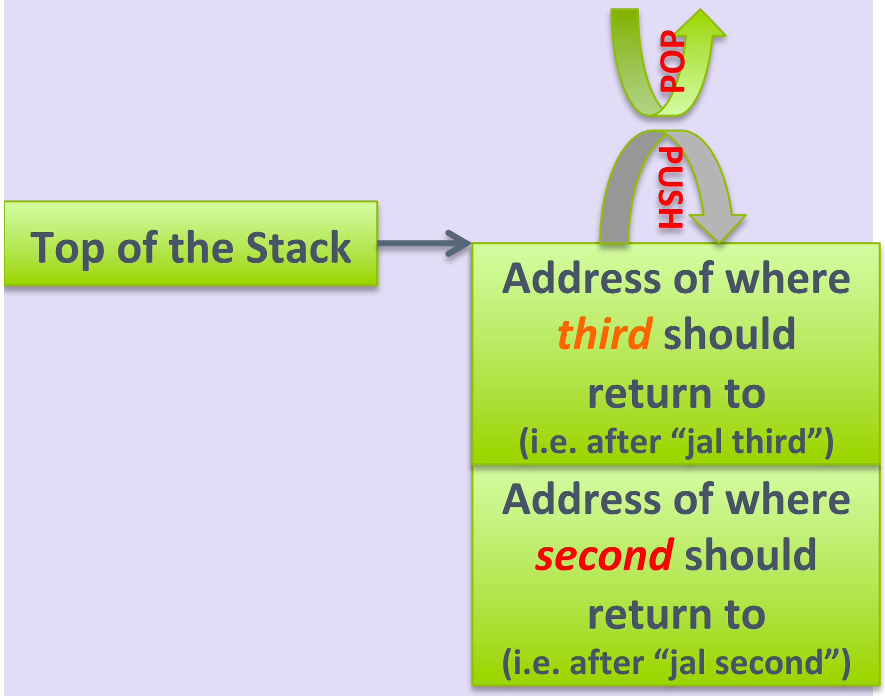
MIPS Call Stack

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

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

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

void forth()
{
   return; }
```




```
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
```

Your To-Dos

- Read the MIPS Calling Convention PDF on the class website!

</LECTURE>