Function Calling Conventions

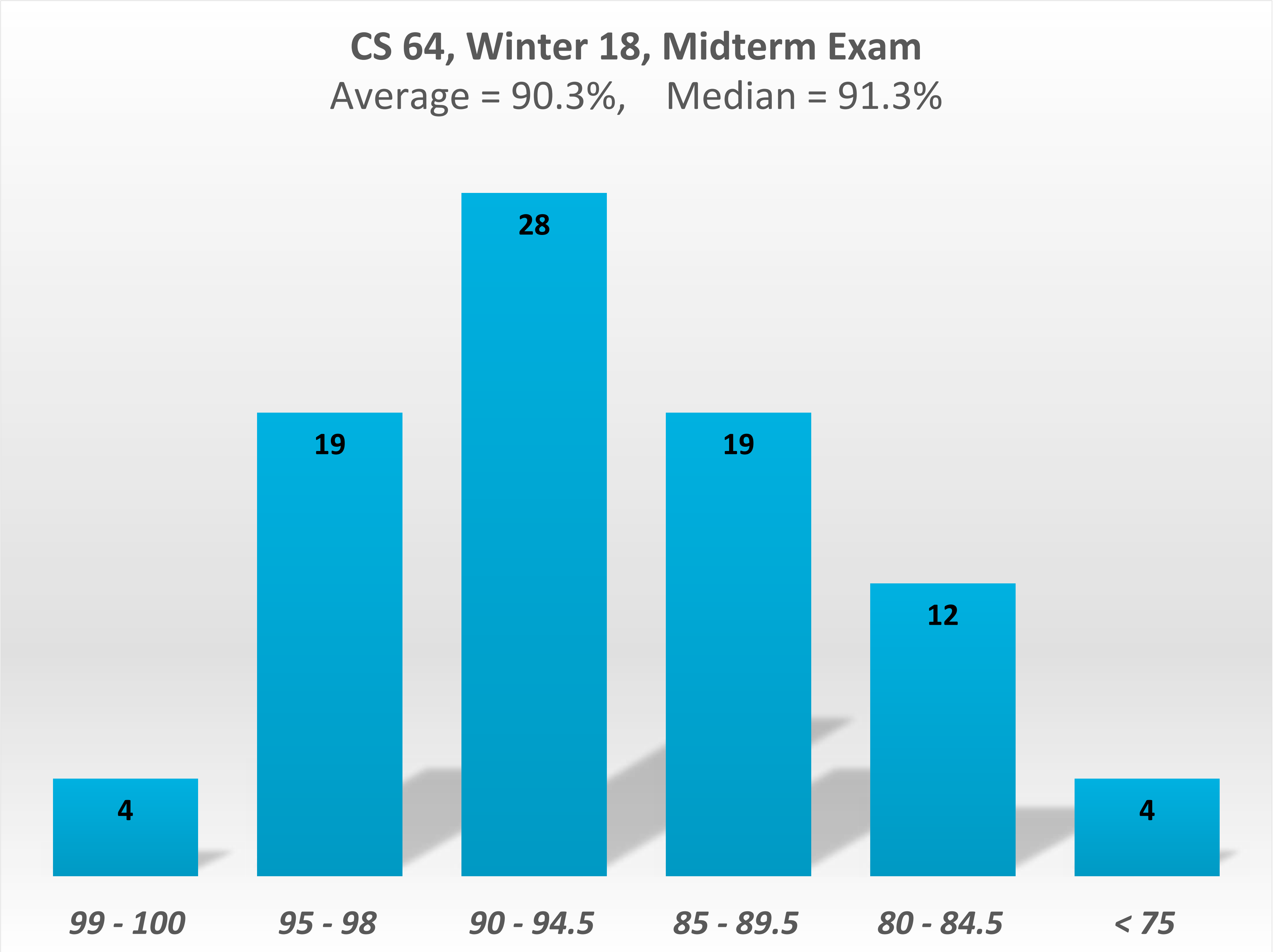
CS 64: Computer Organization and Design Logic
Lecture #10

Ziad Matni
Dept. of Computer Science, UCSB

Administrative

- New assignment this week (Lab 5).
  – Will be posted soon

- Midterm results are in!
  – Posted online
  – Most did quite well – congrats! ☺


CS 64, Winter 18, Midterm Exam
Average = 90.3%,   Median = 91.3%
28
19
19
12
4
4
99 - 100
95 - 98
90 - 94.5
85 - 89.5
80 - 84.5
< 75

Lecture Outline

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

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`**

Simple Call Example

```
.text
test:
    li $a0, 7
    # return to whoever made the call
     jr $ra

main:
     # call test
     jal test

     # print & exit
    li $v0, 1
    syscall
    li $v0, 10
    syscall
```

Passing and Returning Values

- We want to be able to call arbitrary functions without knowing the implementation details

- How might we achieve this?
  - Designate specific registers for **arguments** and **return values**

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**

```
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!
  - Call stacks more than 32 functions deep: ***how common do you think this is?***

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) saves 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)

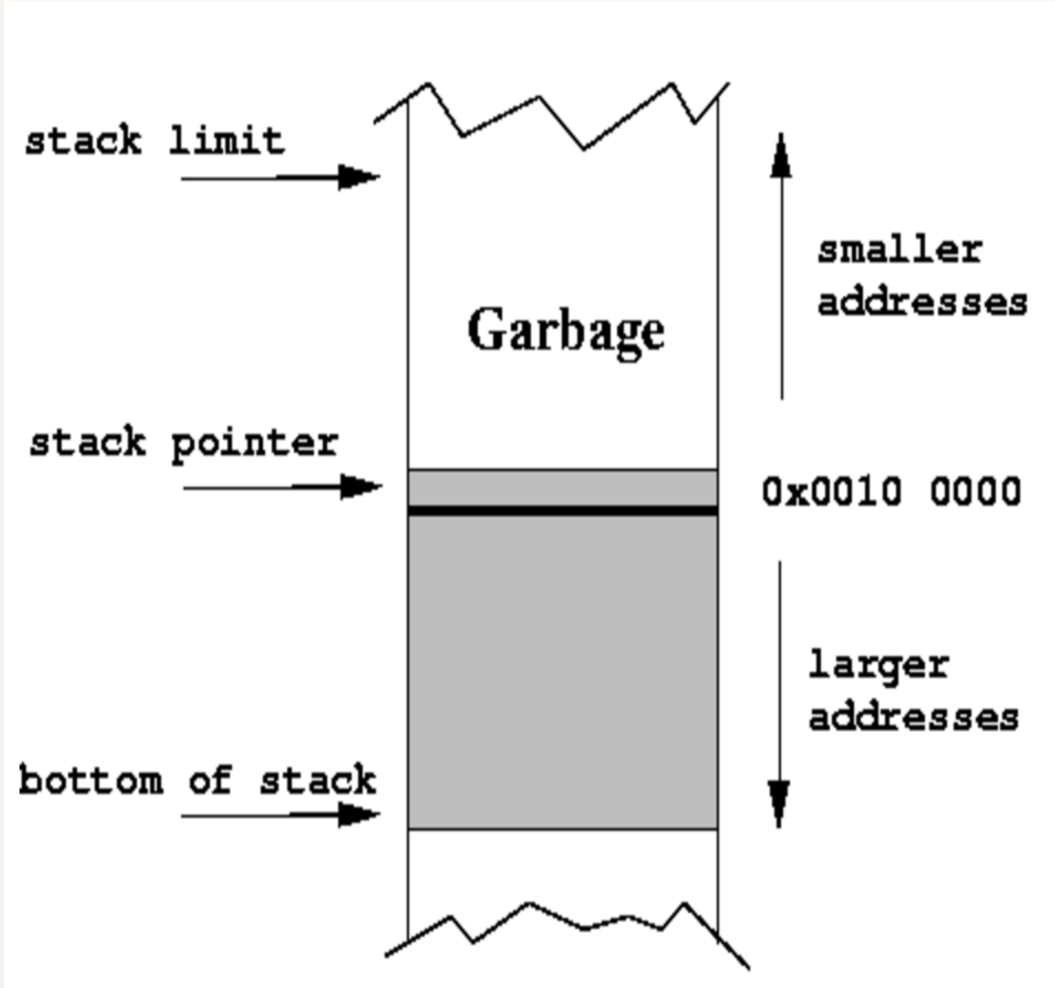
- Non-preserved registers are saved by the ***caller of the function*** (e.g., \$t0 - \$t9)

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

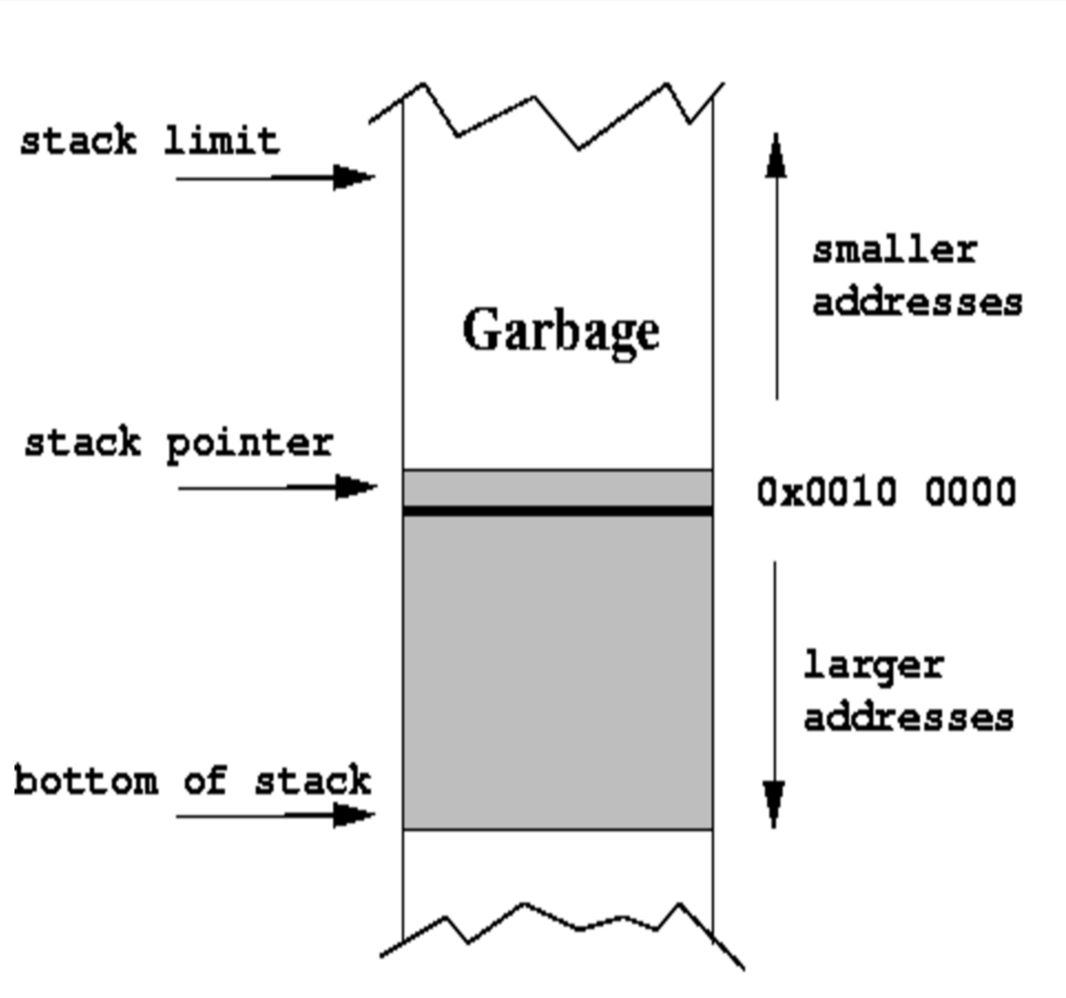
When a program starts executing, a certain *contiguous* section of memory is set aside for the program called the **stack**.



The Stack

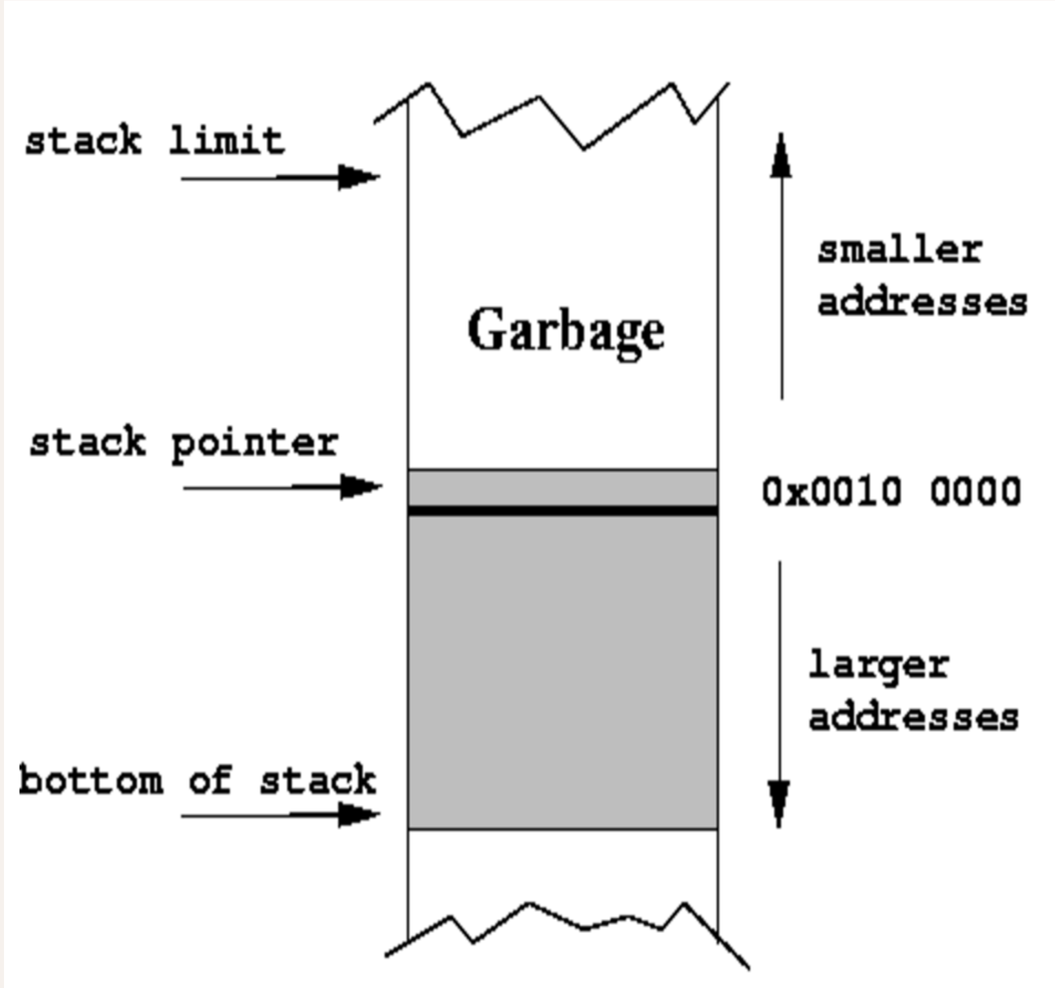
- The **stack pointer** is a register (`$sp`) that contains the **top of the stack**.

- `$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***.



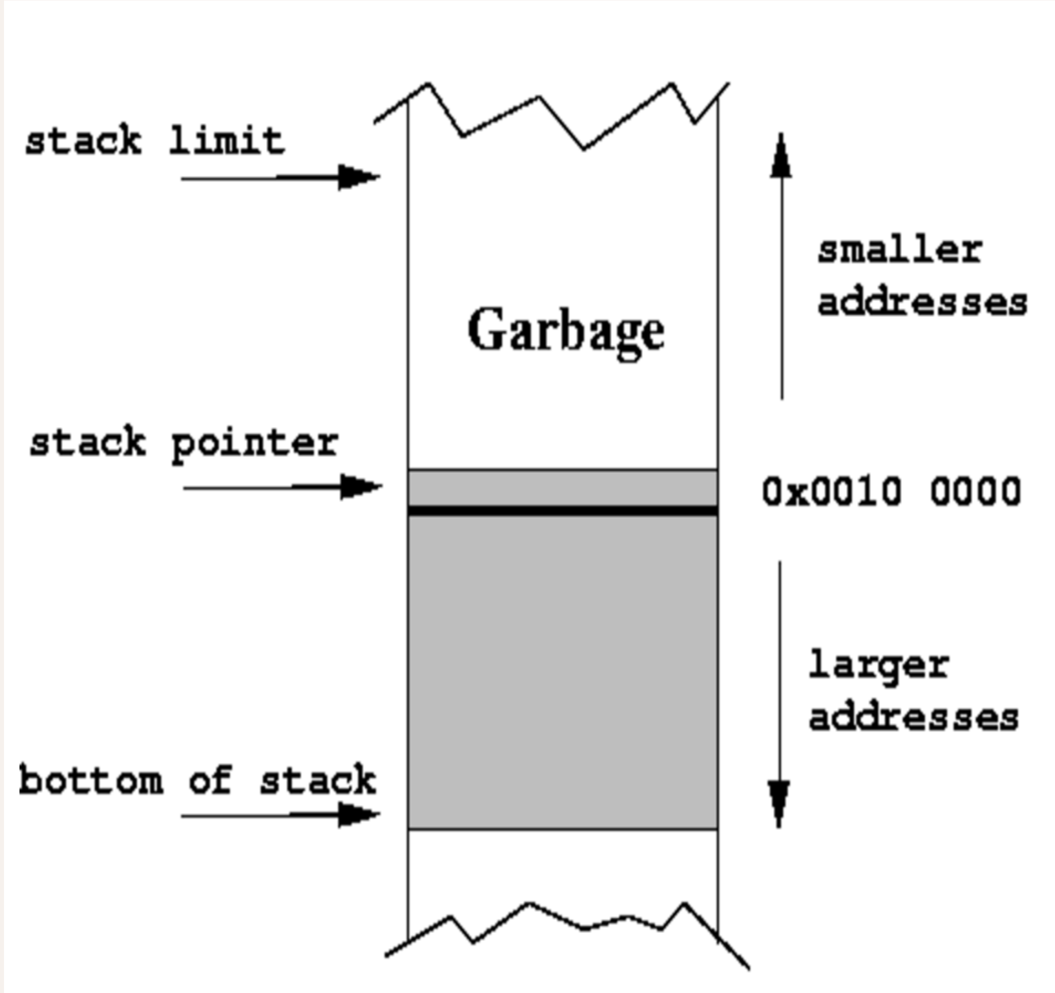
The Stack

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

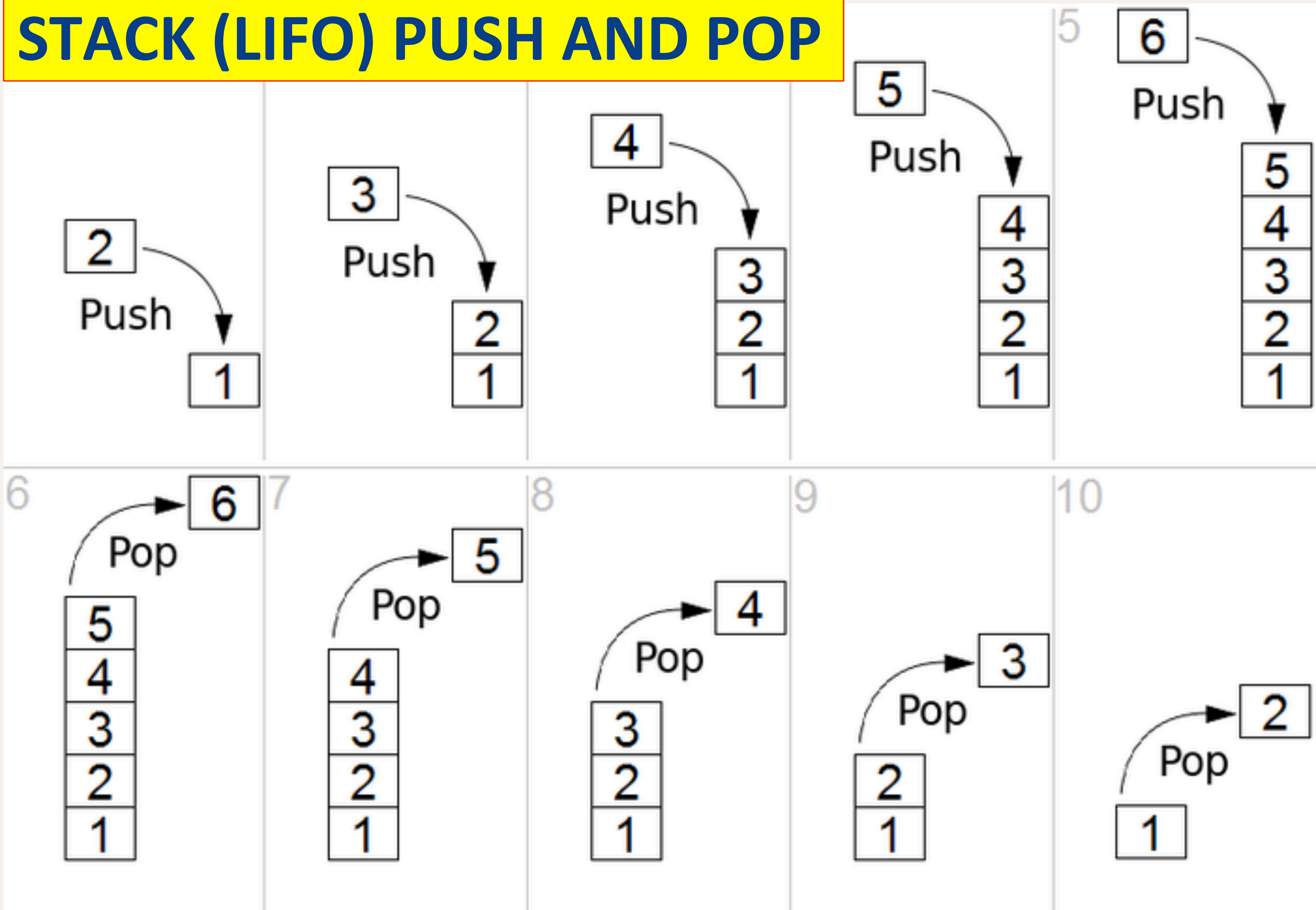


The Stack

- **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
    - Why????
  - Copy the registers *to* the stack (do a **`sw`** instruction)

- **POP** one or more registers
  - Copying 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

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
- **DEMO**: save_registers.asm

save_registers.asm

- The program will look at 2 integers (a0, a1) and ultimately returns (a0 + a0) + (a1 + a1) via a call
- It will create room for **2 words** on the stack
  - It will **push** – i.e. subtract `$sp` & save – 2 words: for **`$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** – i.e. load & add `$sp` – 2 words from the stack & place them in **`$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
        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
        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 in this example
    li $t0, 1

    # 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
    # responsibility to store it on the stack and restore it after add_ints finishes
    addi $sp, $sp, -4
    sw $t0, 0($sp)

    # setup the call and make it
    li $a0, 3
    li $a1, 7
    jal add_ints
    # restore $t0
    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
    la $a0, newline
    li $v0, 4
    syscall
    # print out the prompt for the saved value
    la $a0, saved_text
    li $v0, 4
    syscall
    # print out the saved value
    move $a0, $t0
    li $v0, 1
    syscall
    # exit
    li $v0, 10
    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

- Why do ***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
  - Also, when you are asked by me to use this C.C. and you don't, **you will get no credit** on the assignment or test question

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 is 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 **`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

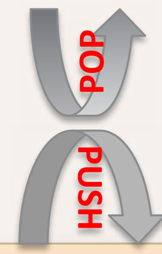
MIPS Call Stack

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

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

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

void forth()
{
   return; }
```



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
```

Functions that Call Functions That Need Additional Registers

- Consider this program:

```
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;
}
```

- 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, according to the MIPS C.C. Rules, ***`$t regs`*** *are* ***<u>unpreserved</u>***

  Also, because we're supposed to use ***$v0*** *for returned values*

- So, what do we do when we have **$t** registers that now need to be used from one function to another??

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`), **are not preserved across calls**.

- You must always assume that unpreserved registers **can/will change values across calls**, even if you know for a fact 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)**

YOUR TO-DOs

- Make sure you’re done reading the handout on MIPS Calling Conventions

</LECTURE>