MIPS Functions

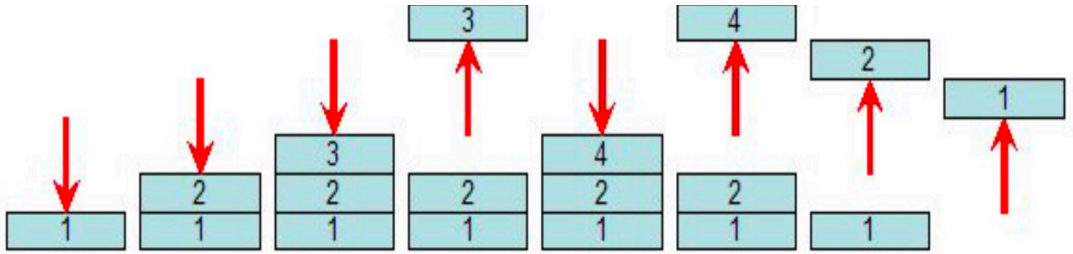
CS 64: Computer Organization and Design Logic

Lecture #9

Winter 2020

Ziad Matni, Ph.D.

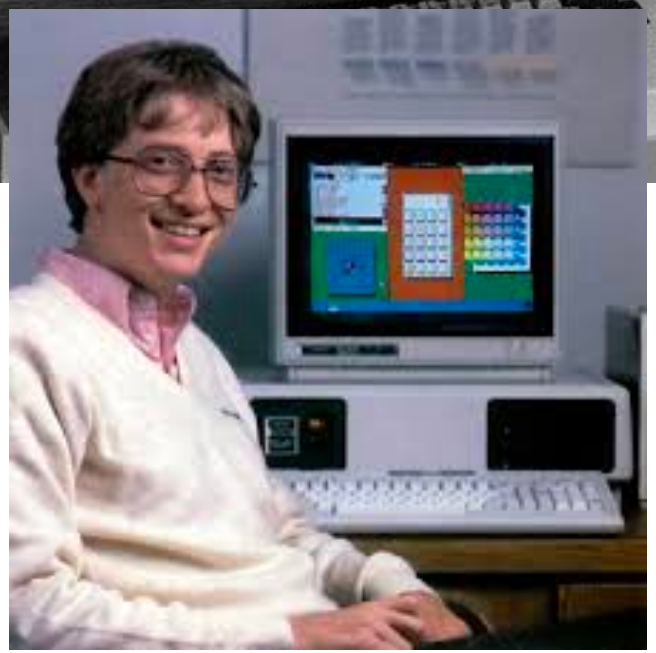
Dept. of Computer Science, UCSB



This Week on
"Didja Know Dat?!"

Xerox PARC (the research arm of the main company) invented the **first GUI** in the **early 1970s** and developed the Alto Computer to show it off, along with the **first mouse** input device AND the **first Ethernet** communication port, but Xerox thought it was all useless (how could those things sell copy machines??)

Steve Jobs and his *frenemy* **Bill Gates** took a tour of Xerox PARC in the early 80s, looking for new ideas. They were shown all of this and were told Xerox wouldn't care much if anyone used the tech!



Right away, Jobs went on to invent the **Macintosh Computer** (which had the first ever commercial GUI-based OS + mouse) & Gates went on to develop **Windows OS** (which quickly overtook Mac OS sales).

Moral of the Story? Don't be shortsighted like Xerox…

Administrative

- Midterm is next week: **Wed. Feb. 12th**
- I'll post practice questions by Friday

- Q:
  Is there a new lab coming up (given the midterm)?
- **A:**
  YES! Will be put up tonight!

What's on the Midterm?

What's on It?

- Everything we've done so far from start thru Monday, 2/10

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**
- I will have some of you re-seated
- Bring your UCSB ID

Lecture Outline

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

Any Questions From Last Lecture?

Functions

- Up until this point, we have not discussed **functions**

- Why not?
  - If you want to do functions, you need to use **the stack**
  - Memory management is a must for the call stack ...
    though we can make *some* progress without it

- Think of recursion…
  - How many variables are we going to need ahead of time?
  - What memory do we end up using in recursive functions?
  - We don't always know…

Implementing Functions

What capabilities do we need for functions?

1. Ability to execute code elsewhere
   - Branches and jumps

2. Way to pass arguments in and out of the function
   - There's a way (a.k.a a convention) to do that that we'll learn about
   - We'll use the registers to do function I/O

Jumping to Code

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

```
void bar() {
  ...
}
```

```
void baz() {
  ...
}
```

- We have ways to jump unconditionally to code (`j` instruction)

- But what about ***jumping back***?
  - That is, after you're done with a function?
  - We'll need a way to *save* where we were (so we can "jump" back)

- **Q**: What do need so that we can do this on MIPS?
  - **A**: A way to store the program counter (**$PC**) multiple times
    (to tell us where the *next* instruction is so that we know *where* to return!)

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

- 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**"*

Passing and Returning Values

- We want to be able to call arbitrary functions without knowing the implementation details
- So, we need to know our pre-/post-conditions
- Q: How might we achieve this in MIPS?
  - A: We 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

- What if we want to pass >4 args?
  - There are ways around that…
    but we won't discuss them in CS64…!

Function Calls Within Functions…

Given what we've said so far…

- What about this code makes our previously discussed setup *break*?
  - ***ANS***: You would need **multiple copies of `$ra`**

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

- You'd have to copy the value of `$ra` somewhere 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!

- 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 are there rules for how to do this?

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

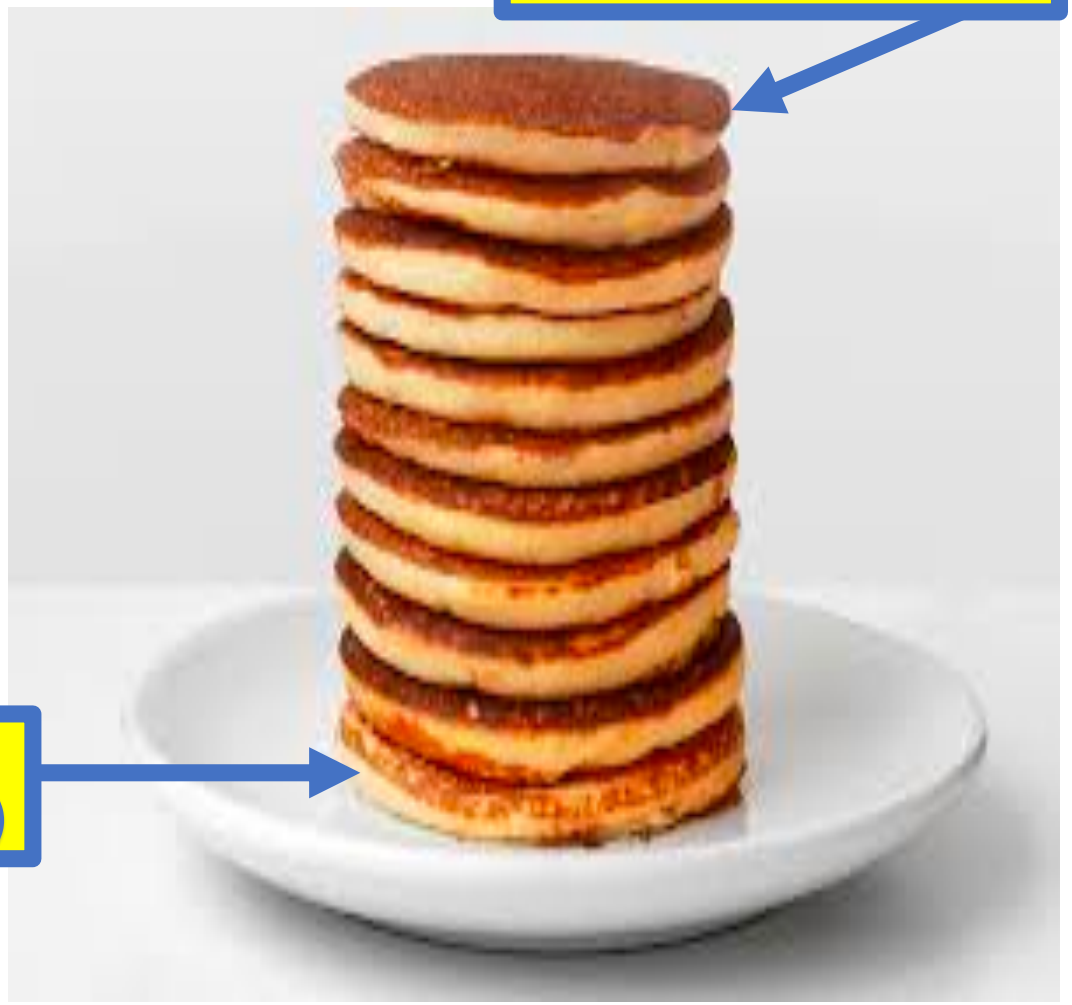
- Non-preserved registers are saved by the ***caller of the function*** (e.g., $t, $a, $v regs)
  - So these should be saved by the function's caller IF THAT IS NEEDED
  - Or not… (the usual route)

And Where is it Saved?

- Register values are saved on the **stack**
- The top of the stack is held in **`$sp`** (**stackpointer**)
- Weirdness of MIPS (and other CPUs):
  The stack grows
  from high addresses *to* low addresses

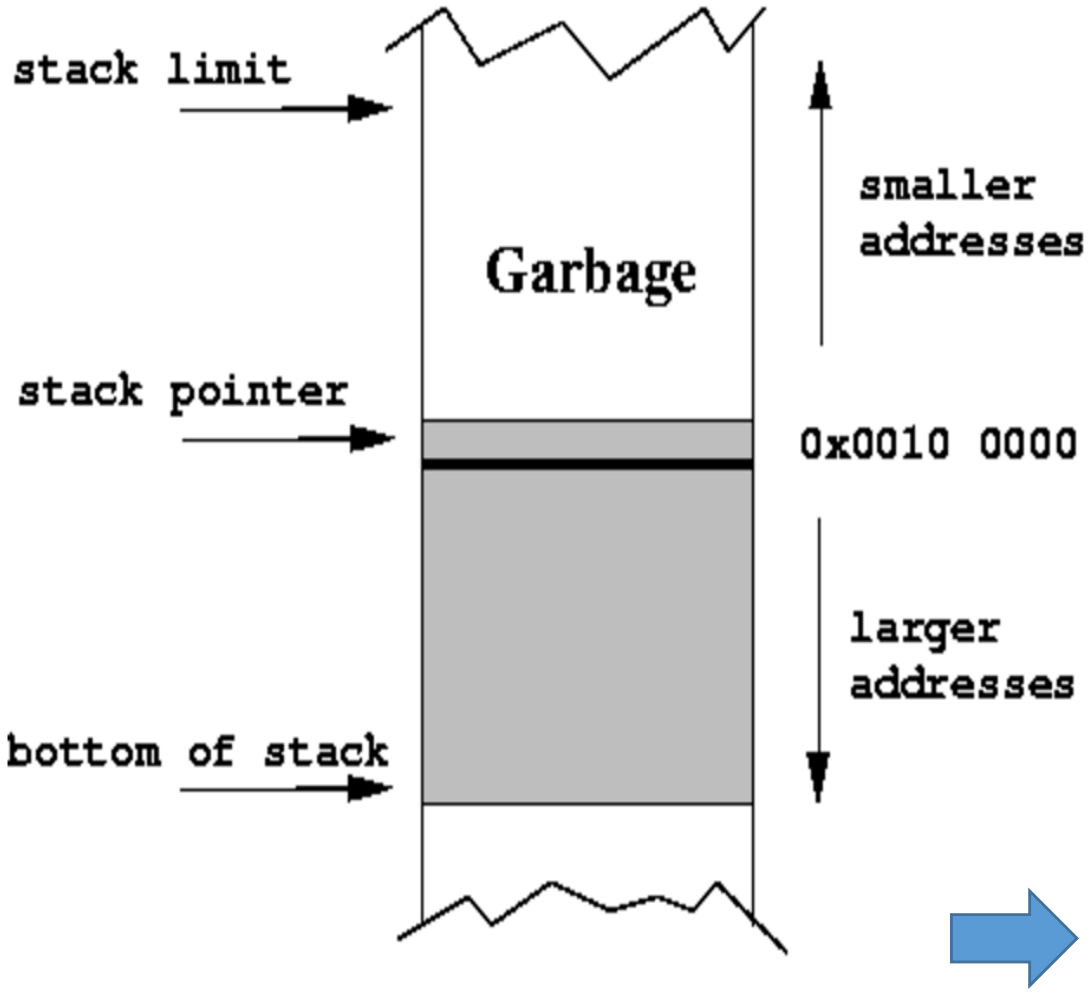
The Stack

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



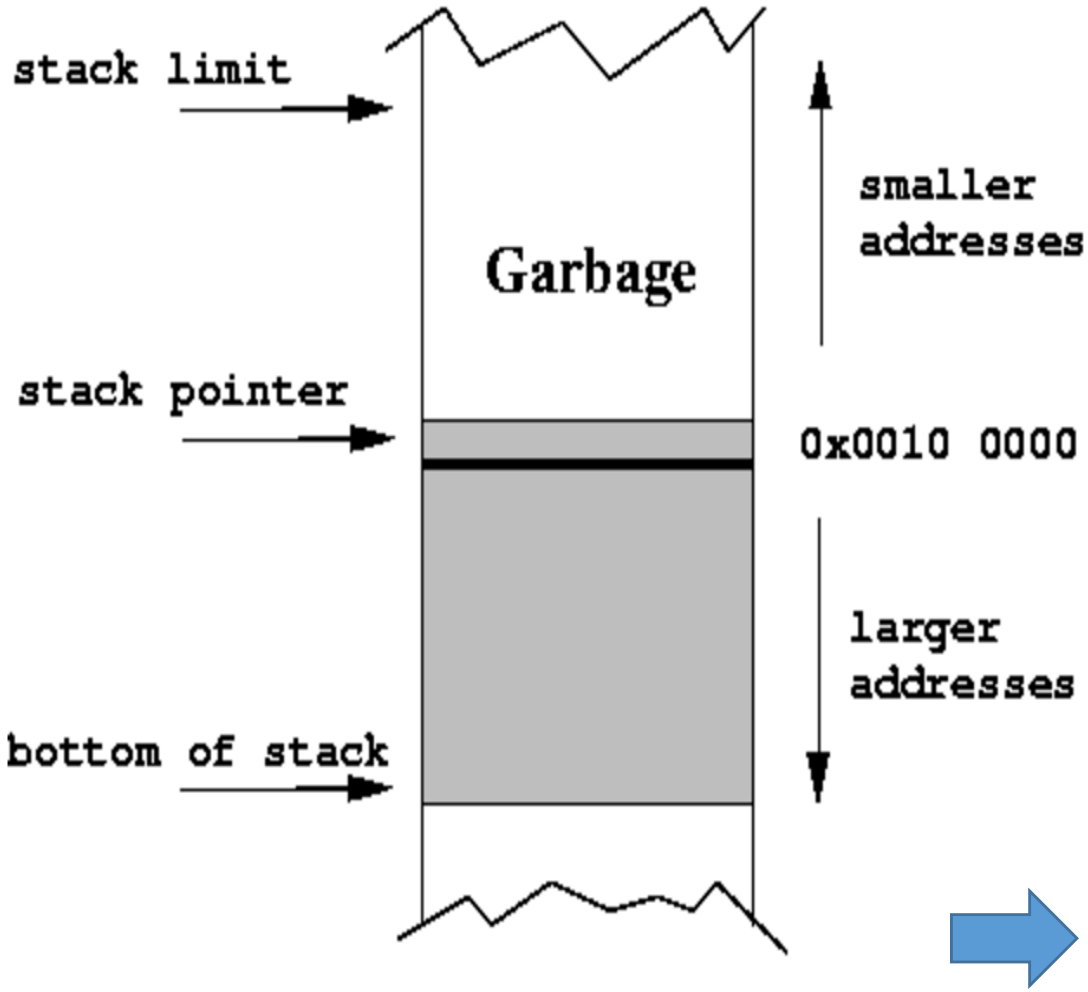
The Stack

- 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***.



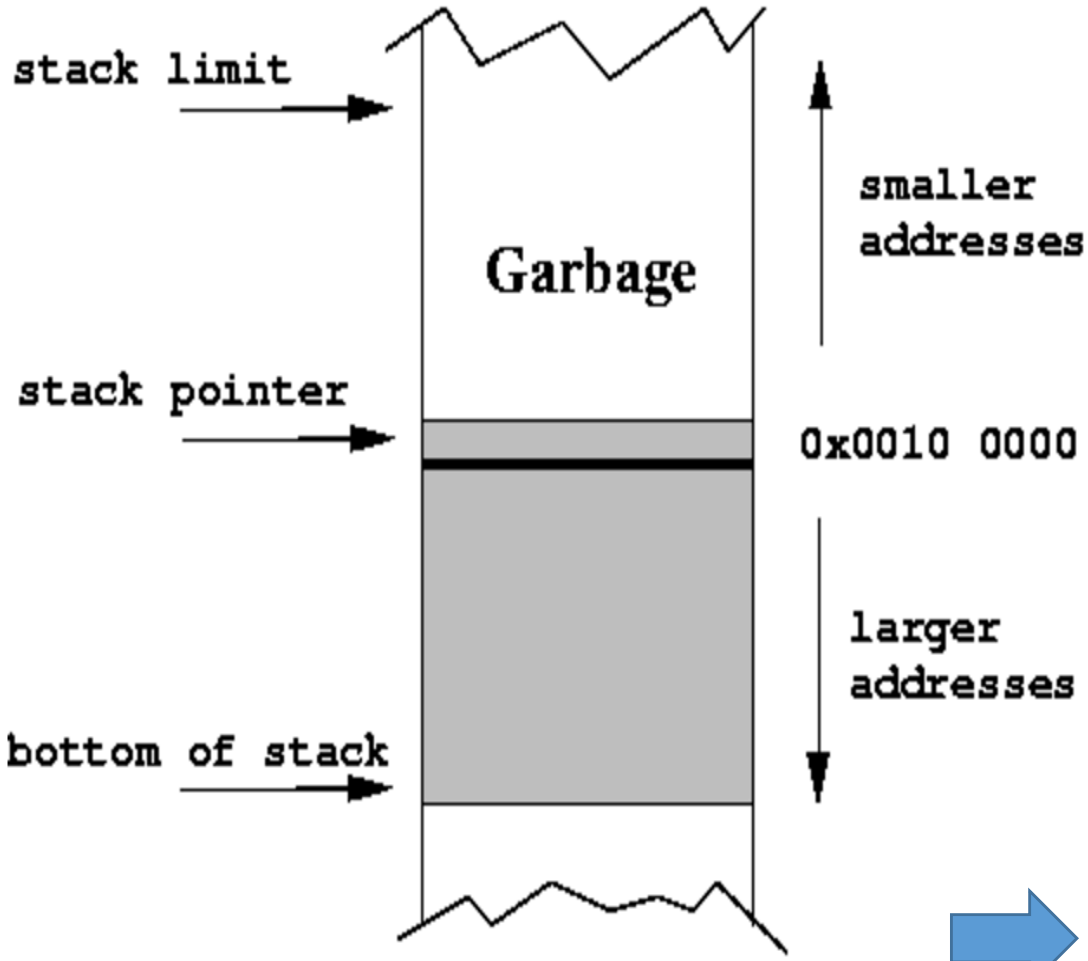
The Stack

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

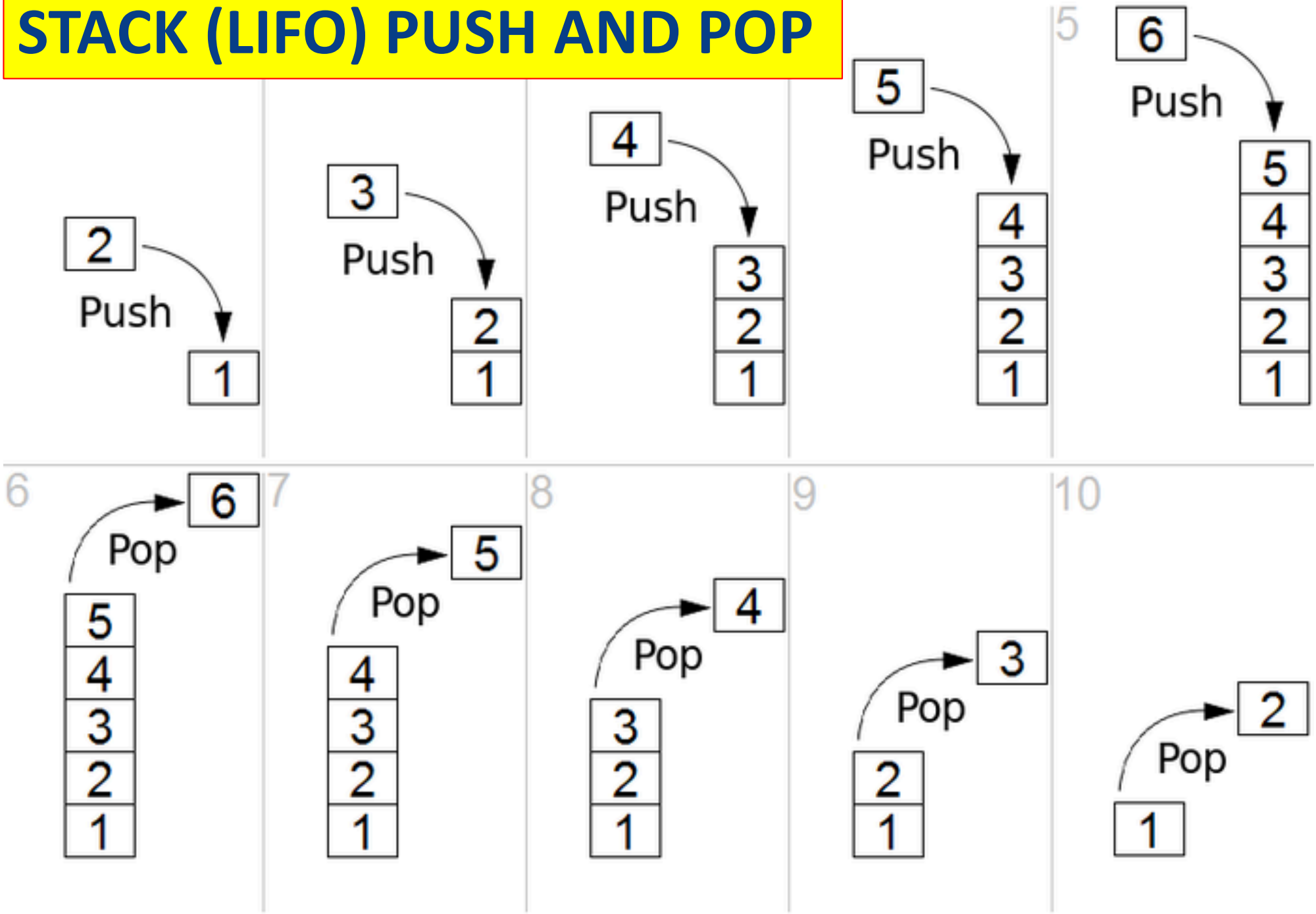


The Stack

- **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
  - Subtract **4 times the number of registers to be pushed** on the stack pointer
    - *Why????*

- Let's say we want to store 2 registers' data into the stack…
- 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

- 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 t-registers are assumed to not be preserved, we can modify them **and it will not
matter**
         li $t0, 4242
         li $t3, -12345678

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

    # print out the solution prompt
    move $t1, $v0      # First, save what's on $v0!!! (why???)
    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

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…

YOUR TO-DOs

<u>IMPORTANT:</u>

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

- Review ALL the demo codes
  - Available via the class website
- Work on Lab #5
- Start studying for the Midterm Exam! ☺


</LECTURE>