More on Memory in MIPS
The Stack and MIPS Calling Convention

CS 64: Computer Organization and Design Logic
Lecture #9

Ziad Matni
Dept. of Computer Science, UCSB
Grade Distribution - CS64 Midterm 1

Mean = 83.02  Median = 86.3
Lecture Outline

• Memory in MIPS
  – With examples for memory access and array vars

• The Stack, featuring Push and Pop!

• MIPS Calling Convention
Arrays

• Question:
  As far as memory is concerned, what is the *major difference* between an **array** and a **global variable**?
  – Arrays contain multiple elements

• Let’s take a look at:
  • print_array1.asm
  • print_array2.asm
  • print_array3.asm
int myArray[]
    = {5, 32, 87, 95, 286, 386}
int myArrayLength = 6
int x

for (x = 0; x < myArrayLength; x++) {
    print(myArray[x])
    print("\n")
}
int myArray[]
    = {5, 32, 87, 95, 286, 386}
int myArrayLength = 6
int* p

for (p = myArray; p < myArray + myArrayLength; p++) {
    print(*p)
    print("\n")
}
MIPS CALLING CONVENTION
Functions

• Up until this point, we have not discussed functions

• Why not?
  – Memory is a must for the call stack ...though we can make some progress without it

• Think of recursion...
Implementing Functions

What capabilities do we need for functions?

• Ability to execute code elsewhere
  – Branches and jumps

• Way to pass arguments
• Way to return values
  – Registers
Jumping to Code

• We have ways to jump to code
• But what about jumping back?
  – Need a way to save where we were

• Q: What do need so that we can do this on MIPS?
  – A: A way to store the program counter
    (to tell us where the next instruction is
    so that we know where to return!)

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

void bar() {
    ...
}

void baz() {
    ...
}
```
Calling Functions on MIPS

• Two crucial instructions: jal and jr

• 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

```assembly
# Calls a function (test) which immediately returns
.text

test:
    # return to whoever made the call
    jr $ra

main:
    # call test
    jal test

    # exit
    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 - $a3
  – Argument registers, for passing function arguments

• Registers $v0, $v1
  – Return registers, for passing return values
Passing and Returning Values in MIPS

Demos

• print_int.asm
  – Illustrates the use of a printing sub-routine
    (i.e. like a simple function)

• add_ints.asm
  – Illustrates the use of an adding sub-routine
    (i.e. like a simple function that returns a value)
Function Calls Within Functions...

Given what we’ve said so far...

• What about this code makes our setup break?
  – You would need **multiple copies of $ra**

• You’d have to copy the value of $ra to another register before calling another function
• You could run out of registers!
• Call stacks more than 32 functions deep: **common or not?**

```c
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?

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

• Store certain information in memory 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

Push 2
Push 3
Push 4
Push 5
Push 6
Pop 6
Pop 5
Pop 4
Pop 3
Pop 2
Pop 1
Stack Push and Pop

- **PUSH** one or more registers
  - Subtract *4 times the number of registers to be pushed* on the stack
  - 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 (stack pointer)

• 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

- Create room for 2 words on the stack
  - Push – i.e. subtract $sp & save – 2 words: for $s0 & $s1
  - We’ll use $s0 and $s1
    - b/c we want them to be preserved across a call

- Calculate the returned value and puts the result in $v0

- We will then restore the registers
  - Pop – i.e. load & add $sp – 2 words & place them in $s0 & $s1
Next Week!

• Recursion in MIPS!
  • Yay!
YOUR TO-DOs

• Assignment #4: Due Fr. 2/10

• PLEASE READ:
  MIPS Calling Convention for CS64
  (it’s on the class web page)