

## Pointless Poll

Clap if you like pizza!



## CS Trivia

In "Lecture II: C vs. Java," The Stack played several roles. Which role(s) did The Stack play?

- a. Mr. Space for Local Variables
- b. Mme. Space for the Stack Pointer
- c. Ms. Space for the Return Address
- d. King Space for Global Variables
- e. Dr. Space for Arguments
- f. Lt. Space for World Peace

## CS Trivia Answer

In "Lecture II: C vs. Java," The Stack played several roles. Which role(s) did The Stack play?

- a. Mr. Space for Local Variables  
some local vars are on The Stack
- c. Ms. Space for the Return Address  
save \$ra if calling another function
- e. Dr. Space for Arguments  
use stack for "a4," "a5," "a6," ...

## CS61C - Machine Structures

### Lecture 7 - Procedure Conventions & The Stack

September 20, 2000

David Oppenheimer & Steve Tu

<http://www-inst.eecs.berkeley.edu/~cs61c/>

## Review 1/2

◦ 3 formats of MIPS instructions in binary:

- Op field determines format

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	immediate		

◦ Operands

- Registers: \$0 to \$31 mapped onto \$zero, \$at, \$v\_, \$a\_, \$s\_, \$t\_, \$gp, \$sp, \$fp, \$ra
- Memory: Memory[0], Memory[4], Memory[8], ... , Memory[4294967292]
  - Index is the "address"

## Review 2/2

◦ Big Idea: Stored Program Concept

- Assembly language **instructions encoded as numbers** (machine language)
  - Everything has an address, **even instructions**
  - Pointer in C == Address in MAL
- 3 pools of memory:
- Static: global variables
  - The Heap: dynamically allocated (malloc())
  - **The Stack: some local variables, some arguments, return address**

## Overview

- C Functions
- MIPS Instructions for Procedures
- The Stack
- Administrivia
- Procedure Conventions
- Practice Compilation
- Instructions Potpourri
- Conclusion

7

## C functions

```
main() {
    int i, j, k, m;

    i = mult(j, k); ... ;
    m = mult(i, i); ... ;
}

int mult (int mcand, int mlier){
    int product;

    product = 0;
    while (mlier > 0) {
        product = product + mcand;
        mlier = mlier -1; }
    return product;
}
```

What information must compiler/programmer keep track of?

8

## Function Call Bookkeeping

- Procedure address Labels
  - Return address \$ra
  - Arguments \$a0, \$a1, \$a2, \$a3
  - Return value \$v0, \$v1
  - Local variables \$s0, \$s1, ..., \$s7;
- Most problems above are solved simply by using **register conventions**.

9

## Instruction Support for Functions (1/4)

```
C ... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

```
M address
I 1000 add $a0,$s0,$zero # x = a
P 1004 add $a1,$s1,$zero # y = b
S 1008 addi $ra,$zero,1016 # $ra=1016
1012 j sum #jump to sum
1016 ...

2000 sum: add $v0,$a0,$a1
2004 jr $ra # new instruction
```

10

## Instruction Support for Functions (2/4)

- Single instruction to jump and save return address: jump and link (jal)
- Before:  

```
1008 addi $ra,$zero,1016 # $ra=1016
1012 j sum #go to sum
```
- After:  

```
1012 jal sum # $ra=1016, go to sum
```
- Why have a jal? Make the common case fast: functions are very common.

11

## Instruction Support for Functions (3/4)

- Syntax for jal (jump and link) is same as for j (jump):  

```
jal label
```
- jal should really be called laj for "link and jump":
  - Step 1 (link): Save address of next instruction into \$ra (Why?)
  - Step 2 (jump): Jump to the given label

12

### Instruction Support for Functions (4/4)

- Syntax for jr (jump register):  
jr register
- Instead of providing a label to jump to, the jr instruction provides a register that contains an address to jump to.
- Usually used in conjunction with jal, to jump back to the address that jal stored in \$ra before function call.

13

### Nested Procedures (1/2)

- ```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```
- Something called sumSquare, now sumSquare is calling mult.
  - So there's a value in \$ra that sumSquare wants to jump back to, but this will be overwritten by the call to mult.
  - Need to save sumSquare return address before call to mult.

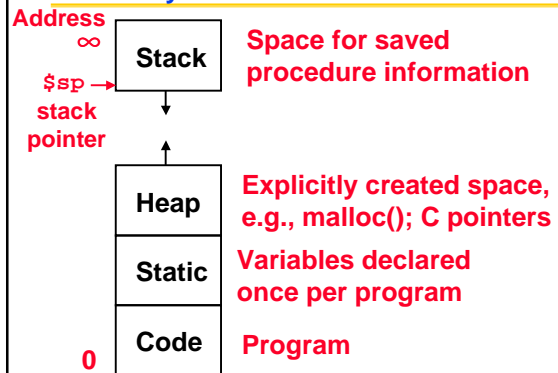
14

### Nested Procedures (2/2)

- In general, may need to save some other info in addition to \$ra.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes
  - Heap: Variables declared dynamically
  - Stack: Space to be used by procedure during execution; this is where we can save register values
    - Not identical to the "stack" data structure!

15

### C memory Allocation



16

### Using the Stack

- So we have a register \$sp which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.

17

### Compiling nested C func into MIPS

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

C }

```
sumSquare:  
Prologue  subi $sp,$sp,12 # space on stack  
          sw $ra,$ 8($sp) # save ret addr  
          sw $a0,$ 0($sp) # save x  
          sw $a1,$ 4($sp) # save y  
          addi $a1,$a0,$zero # mult(x,x)  
Body      jal mult # call mult  
          lw $ra,$ 8($sp) # get ret addr  
          lw $a0,$ 0($sp) # restore x  
          lw $a1,$ 4($sp) # restore y  
Epilogue  add $v0,$v0,$a1 # mult()+y  
          addi $sp,$sp,12 # => stack space  
          jr $ra
```

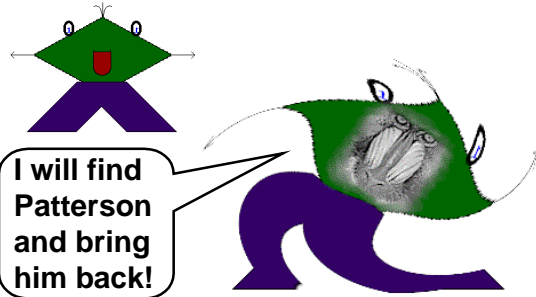
18

### Administrivia (1/2)

- Most assignments are now submitted online (even homeworks)!
- Vote on when Proj2 (sprintf) is due
  - Fri at midnight vs. Sun at midnight
- Lab4 due by the beginning of your discussion this week
- Book has errata.
- See errata on web!
- Errata has errata?

19

### Return of M'Piero!



Summer Mishserghi, Alex Fabrikant  
M'Piero created by David A. Patterson

20

### The Functional Contract

- ❖ Definitions
  - ❖ Caller: function making the call, using jal
  - ❖ Callee: function being called
- ❖ Before the Functional Contract, there was anarchy
  - ❖ Shared registers => Callee overwrote Caller's information
- ❖ Functional Contract established to bring peace to the land
  - ❖ Callee's Rights and Responsibilities
  - ❖ Caller's Rights and Responsibilities

21

### Callee's Responsibilities (how to write a fn)

1. If using \$s or big local structs, slide \$sp down to reserve memory:  
e.g. addi \$sp, \$sp, -48
2. If using \$s, save before using:  
e.g. sw \$s0, 44(\$sp)
3. Receive args in \$a0-3, add'l args on stack
4. Run the procedure body
5. If not void, put return values in \$v0,1
6. If applicable, undo steps 2-1  
e.g. lw \$s0, 44(\$sp)    addi \$sp, \$sp, 48
7. jr \$ra

22

### Compile using pencil and paper! (1/2)

```
int Doh(int i, int j, int k, int m, char c, int n){  
    return i+j+n;  
}
```

Doh: \_\_\_\_\_  
      add \$a0, \_\_\_\_\_  
      \_\_\_\_\_  
      \_\_\_\_\_

23

### Compile using pencil and paper! (2/2)

```
int Doh(int i, int j, int k, int m, char c, int n){  
    return i+j+n;  
}
```

Doh: lw \$t0, 20(\$sp)  
      add \$a0, \$a0, \$a1  
      add \$v0, \$a0, \$t0  
      jr \$ra

VAT  
Safe  
For  
Callee

6th argument

24

### Caller's Responsibilities (how to call a fn)

1. Slide \$sp down to reserve memory: e.g. `addi $sp, $sp, -28`
2. Save \$ra on stack b/c jal clobbers it: e.g. `sw $ra, 24($sp)`
3. If you'll still need their values after the function call, save \$v, \$a, \$t on stack or copy to \$s registers. Callee can overwrite VAT, but not S.
4. Put first 4 words of args in \$a0-3, at most 1 arg per word, add'l args go on stack: "a4" is 16(\$sp)
5. jal to the desired function
6. Receive return values in \$v0, \$v1
7. Undo steps 3-1: e.g. `lw $t0, 20($sp)` `lw $ra, 24($sp)` `addi $sp, $sp, 28`

Keep this slide in mind for exam cheat sheet

25

### Hard Compilation Problem (1/2)

```
int Doh(int i, int j, int k, int m, char c, int n){
    return i+j+n; }
int Sum(int m, int n); /* returns m + n */
```

Doh: \_\_\_\_\_ add \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

jal \_\_\_\_\_

26

### Slow-motion Replay (1/2)

```
int Doh(int i, int j, int k, int m, char c, int n){
    return i+j+n; }
```

```
int Sum(int m, int n); /* returns m + n */
```

|                                      |                               |
|--------------------------------------|-------------------------------|
| Doh: <code>addi \$sp, \$sp, -</code> | add _____                     |
| <code>sw \$ra, (\$sp)</code>         | _____                         |
| _____                                | <code>lw \$ra, (\$sp)</code>  |
| <code>lw \$s0, (\$sp)</code>         | <code>addi \$sp, \$sp,</code> |
| <code>jal Sum</code>                 | <code>jr \$ra</code>          |

|                                  |                                  |                                  |
|----------------------------------|----------------------------------|----------------------------------|
| 1. Calling Sum => save \$ra, VAT | 2. Saving \$ra => must move \$sp | 3. Need n after funccall => \$s0 |
|----------------------------------|----------------------------------|----------------------------------|

27

### Callees' Rights, Callers' Rights

- ❖ **Callees' Rights**
  - ❖ Right to use VAT registers freely
  - ❖ Right to assume args are passed correctly
- ❖ **Callers' Rights**
  - ❖ Right to use S registers without fear of being overwritten by Callee
  - ❖ Right to assume return value will be returned correctly

28

### Legalese

1. Proj2 (sprintf) passes all args on the stack.
2. Floating point has different conventions.
3. Variable # of arguments, specified by ... in C, is slightly different.
4. See manuals for details.

29

### Instructions Potpourri

- 9 new instructions in 9 minutes!
- Multiplication and Division: `mul, mulu, div, divu, mfhi, mflo`
- Accessing Individual Bytes Instead of Words: `lb, lbu, sb`

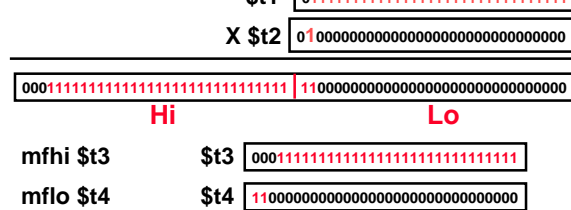
30

### Multiplication

mul \$t1, \$t2 # t1 \* t2

No dest register: Product could be  $\sim 2^{64}$ ; need two special registers to hold it

3-step process



### Division

div \$t1, \$t2 # t1 / t2

Quotient stored in Lo

Bonus prize: Remainder stored in Hi

mflo \$t3 #copy quotient to t3

mfhi \$t4 #copy remainder to t4

3-step process

### Unsigned Multiplication and Division

mulu \$t1, \$t2 # t1 \* t2

divu \$t1, \$t2 # t1 / t2

Just like mul, div, except now interpret t1, t2 as unsigned integers instead of signed

Answers are also unsigned, use mfhi, mflo to access

### Data Types in MAL

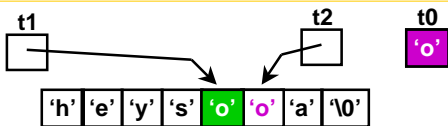
What if t1, t2 are signed ints, and you try to do mulu, divu?

- a) Segmentation fault?
- b) Bus error?
- c) Green gecko?

**NO! None of the above!**

**BIG IDEA: registers contain TYPELESS, MEANINGLESS BIT PATTERNS!**  
Signed/unsigned/char/color is determined by instruction or operation

### Load byte, store byte

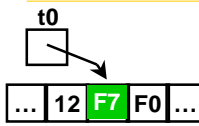


lb \$t0, 0(\$t1)

sb \$t0, 0(\$t2)

Similar to lw, sw, except bytes instead of words

### Load byte unsigned



lb \$t1, 0(\$t0) FFFFFFF7 Sign-extended

lbu \$t2, 0(\$t0) 00000F7 Zero-extended

## Big Ideas

- Follow the procedure conventions and nobody gets hurt.
- Data is just 1's and 0's, what it represents depends on what you do with it
- M'Piero has returned to find Prof. Patterson, if he still lives!

37

## Summary of Instructions & Registers

- Registers we know so far
  - \$0, \$at, \$ra, \$v\_, \$a\_, \$t\_, \$s\_, \$gp, \$sp
- Instructions we know so far
  - Arithmetic: add, addu, addi, addiu, sub, subu, mul, mulu, div, divu, mflo, mfhi
  - Memory: lw, sw, lb, lbu, sb
  - Decision/Comparison: beq, bne, slt, sltu, slti, sltiu
  - Unconditional Branches (Jumps): j, jal, jr

38