MIPS Procedure Calls

Lecture 6
CS301
Function Call Steps

- Place parameters in accessible location
- Transfer control to function
- Acquire storage for procedure variables
- Perform calculations in function
- Place result in place accessible to caller
- Return control to caller
MIPS Function Calls

- **Parameters**
  - $a0–a3$
- **Return values**
  - $v0–v1$
- **Where to return control to**
  - $ra$
MIPS Function Calls

• Transfer control to function
  ◦ jal label
    ▪ Jumps to label’s instruction
    ▪ Stores return address in $ra (PC+4)

• Return control to caller
  ◦ jr $ra
Other Register Conventions

• Caller–saved registers
  • $t0–$t9, $a0–$a3
  • No preservation assumed

• Callee–saved registers
  • $s0–$s7
  • If you use these, you must restore values before returning

• Stack pointer
  • $sp
  • Points to last location on stack
Examples

- Suppose $s0 = a$, $s1 = b$, $s2 = c$, $s3 = d$
- Write MIPS instructions for the following code (assuming code for ABS already written):

  \[ b = \text{ABS}(d) \]
Address Space

- Each process has an address space
- The address space is divided into segments:
  - Text
    - Instructions
  - Initialized Data
    - Globals
  - Uninitialized Data or Heap
    - `new` allocates space here
  - Stack
    - local variables are given space here
MIPS Function Calls: Local Storage

- **Stack**
  - LIFO
  - `$sp`
- **Non-volatile registers**
  - Push onto stack at function call
  - Restore to registers before function return
- **Spill local register values onto stack if not enough registers for function operation**
Procedure Frame/Activation Record

- Segment of stack that contains procedure’s saved registers and local variables
- Frame pointer ($fp) points to first word of procedure frame
Procedure Frame/ Activation Record

- Segment of stack that contains procedure’s saved registers and local variables
- Frame pointer ($fp$) points to first word of procedure frame (a.k.a. stack frame)

fp a.k.a. base pointer (bp).
Register 30 (s8) in MIPS.
- Segment of stack that contains procedure’s saved registers and local variables
- Frame pointer ($fp$) points to first word of procedure frame (sort of)
int CalculateTriangleArea(int b, int h)
{
    int area = b * h;
    area /= 2;
    return area;
}

int main()
{
    int b = 4;
    int h = 10;
    int val = CalculateTriangleArea(b, h);
}
About the code that follows...

- It was generated by a compiler, so it’s not like code one would write.
- Some assemblers use $s8 to store the frame pointer (this code does).
- $gp (the “global pointer” register), when used, points to a pool of global data that can be commonly referenced by all functions.
  - Convention dictates you should always store it when you code a function (who knows why).
12:   int b = 4;
[  12] 0x100010c0:  24 02 00 04  li v0,4
[  12] 0x100010c4:  af c2 00 10  sw v0,16(s8)
13:   int h = 10;
[  13] 0x100010c8:  24 02 00 0a  li v0,10
[  13] 0x100010cc:  af c2 00 14  sw v0,20(s8)
14:   int val = CalculateTriangleArea(b, h);
[  14] 0x100010d0:  8f c4 00 10  lw a0,16(s8)
[  14] 0x100010d4:  8f c5 00 14  lw a1,20(s8)
[  14] 0x100010d8:  8f 99 80 68  lw t9,-32664(gp)
[  14] 0x100010dc:  03 20 f8 09  jalr t9
[  14] 0x100010e0:  00 00 00 00  nop
[  14] 0x100010e4:  af c2 00 18  sw v0,24(s8)
Example

```c
int pow(int base, int exponent)
// Assumes base and exponent are both >= 0
{
    int result = 1;
    for(int i = 0; i < exponent; i++){
        result *= base;
    }
    return result;
}
```
Solution

```
addiu $sp, $sp, -4
sw $ra, 0($sp)
li $v0, 1
add $t0, $0, $0
loop: bge $t0, $a1, done
mul $v0, $v0, $a0
addi $t0, $t0, 1
b loop
done: lw $ra, 0($sp)
addiu $sp, $sp, 4
jr $ra
```
Recursive Functions

```c
int fact(int n)
{
    if(n < 1)
        return 1;
    else
        return (n*fact(n-1));
}
```
Recursive Functions

```c
int fact(int n) {
    if(n < 1)  
        return 1;
    else
        return (n*fact(n-1));
}
```

- Acquire storage for procedure variables
- Perform calculations in function
- Place result in place accessible to caller
- Return control to caller
addiu $sp, $sp, -8
sw $ra, 0($sp)
sw $a0, 4($sp)
li $t0, 1
blt $a0, $t0, lessThan
addi $a0, $a0, -1
jal fact
lw $a0, 4($sp)
mul $v0, $v0, $a0
lw $ra, 0($sp)
addiu $sp, $sp, 8
jr $ra
lessThan: li $v0, 1
lw $ra, 0($sp)
addiu $sp, $sp, 8
jr $ra
int pow(int base, int exponent)
// Assumes base and exponent are both >= 0
{
    int result = 1;

    if(exponent == 0)
        return result;
    else{
        result = base * pow(base, exponent-1);
    }
    return result;
}
bne $a1, $0, else
add $v0, $v0, $0
jr $ra
else: addiu $sp, $sp, -8
sw $ra, 0($sp)
sw $a0, 4($sp)
addi $a1, $a1, -1
jal pow
lw $a0, 4($sp)
mul $v0, $v0, $a0
lw $ra, 0($sp)
addiu $sp, $sp, 8
jr $ra
System Calls

- Used to interact with operating system
- For our purposes, use for I/O
  - Print output to console
- syscall
  - Place arguments to syscall in registers
  - Put number specifying which syscall into $v0
  - It’s like a function call with respect to register conventions

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0=integer</td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0=string</td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td>result in $v0</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0=buffer, $a1=length</td>
</tr>
</tbody>
</table>
Given the following function header, int foo(int a, int b); what will be on the stack before any of the calculations in foo are performed? Assume foo() calls some other function.
What will be on the stack on a call to
int foo(int a, int b, int c, int d, int e, int f)?