Upcoming Calendar

- Lab Due Dates
  - 10/16, 10/30, 11/13, 12/4
- Homework Due Dates
  - 10/23, 11/6, 11/20, 12/11
- Midterm
  - 10/30
- Class Canceled
  - 11/25

Review

- What is the problem concerning saving registers across function calls in assembly language?
- Who saves $a0?
- Who saves $t0?
- Who saves $s0?
- Who saves $ra?
Where are the registers saved?

- Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.
- It would be nice if each function call had its own private memory area.
  - This would prevent other function calls from overwriting our saved registers—otherwise using memory is no better than using registers.
  - We could use this private memory for other purposes too, like storing local variables.

Function calls and stacks

- Notice function calls and returns occur in a stack-like order: the most recently called function is the first one to return.

1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns

- Here, for example, C must return to B before B can return to A.
Stacks and function calls

- It’s natural to use a stack for function call storage. A block of stack space, called a stack frame, can be allocated for each function call.
  - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
  - Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- The stack frame can be used for several purposes.
  - Caller- and callee-save registers can be put in the stack.
  - The stack frame can also hold local variables, or extra arguments and return values.

The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
  - The stack grows downward in terms of memory addresses.
  - The address of the top element of the stack is stored (by convention) in the “stack pointer” register, $sp.
- MIPS does not provide “push” and “pop” instructions. Instead, they must be done explicitly by the programmer.
Pushing elements

- To push elements onto the stack:
  - Move the stack pointer $sp$ down to make room for the new data.
  - Store the elements into the stack.

- For example, to push registers $t1$ and $t2$ onto the stack:
  
  ```
  addi $sp, $sp, -8  
  sw $t1, 4($sp)  
  sw $t2, 0($sp)
  ```

- An equivalent sequence is:
  
  ```
  sw $t1, -4($sp)  
  sw $t2, -8($sp)  
  addi $sp, $sp, -8
  ```

Accessing and popping elements

- Any element in the stack can be referenced if you know where it is relative to $sp$.

- For example, to retrieve the value of $t1$:
  
  ```
  lw $s0, 4($sp)
  ```

- Pop, or “erase,” elements by adjusting the stack pointer upwards

- To pop the value of $t2$, yielding the stack shown at the bottom:
  
  ```
  addi $sp, $sp, 4
  ```

- Popped data is still present in memory, but data past the stack pointer is considered invalid.
Representing Strings

- C-style string is represented by an array of bytes
  - Elements are 1-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>97</td>
<td>114</td>
<td>114</td>
<td>121</td>
<td>32</td>
<td>111</td>
<td>116</td>
<td>116</td>
<td>101</td>
<td>114</td>
<td>0</td>
</tr>
<tr>
<td>space</td>
<td>!</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>,</td>
<td>.</td>
<td>/</td>
</tr>
</tbody>
</table>

strlen Example

```c
void somefunc() {
    char *str; int a;
    ...
    /* uses t0, t1 somewhere */
    ...
    a = strlen(str);
    ...
}

int strlen(char *s) {
    int count = 0;
    while (*s != 0) {
        count++;
        s++;
    }
    return count;
}
```

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

```
addi $sp, $sp, -12
sw $ra, 8($sp)
sw $t0, 4($sp)
sw $t1, 0($sp)
add $a0, $t0, $0
jal strlen
lw $t1, 0($sp)
lw $t0, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
...
jr $ra
```

last_sf_word

last_sf_word

val of $ra

val of $t0

val of $t1

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra
void somefunc() {
    char *str; int a;
    /*uses t0, t1 somewhere*/
    a = strlen(str);
}

int strlen(char *s) {
    int count = 0;
    while (*s != 0) {
        count++;
        s++;
    }
    return count;
}

somefunc:
    ...  
addi $sp, $sp, -12
sw $ra, 8($sp)
sw $t0, 4($sp)
sw $t1, 0($sp)
add $a0, $t0, $0
jal strlen
lw $t1, 0($sp)
lw $t0, 4($sp)
add $ra, 8($sp)
addi $sp, $sp, 12
;jr $ra

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

int strlen(char *s) {
    int count = 0;
    while (*s != 0) {
        count++;
        s++;
    }
    return count;
}

somefunc:
    ...  
addi $sp, $sp, -12
sw $ra, 8($sp)
sw $t0, 4($sp)
sw $t1, 0($sp)
add $a0, $t0, $0
jal strlen
lw $t1, 0($sp)
lw $t0, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
;jr $ra

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra
Frame Pointer and Reference

```c
void somefunc() {
  char *str; int a;
  ...
  /*uses t0, t1 somewhere*/
  a = strlen(str);
  ...
}
```

```asm
somefunc:
  ...
  addi $sp, $sp, -12
  sw $ra, 8($sp)
  sw $t0, 4($sp)
  sw $t1, 0($sp)
  add $a0, $t0, $0
  jal strlen
  lw $t1, 0($sp)
  lw $t0, 4($sp)
  lw $ra, 8($sp)
  addi $sp, $sp, 12
  ...
  jr $ra
```

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

---

Heavyweight Fcns – Set Up Frame

```c
void somefunc() {
  int a, b, c;
  ...
  a = b + c;
  ...
}
```

```asm
somefunc:
  ...
  addi $sp, $sp, -48
  sw $s0, 44($sp)
  sw $s1, 40($sp)
  ...
  sw $s7, 16($sp)
  sw $ra, 12($sp)
  sw $0, 8($sp) Initialize a $sp
  sw $0, 4($sp) Initialize b
  sw $0, 0($sp) Initialize c
  move $fp, $sp
  ...
```

caller-saved: $s0-$s9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra
**Heavyweight Fcns – Tear Down Frame**

```c
void somefunc() {
    int a, b, c;
    ...
    a = b + c;
    ...
}
```

```assembly
sw $s0, 44($sp)
sw $s1, 40($sp)
...
sw $s7, 16($sp)
sw $ra, 12($sp)
addi $sp, $sp, 48
move $fp, $sp
jr $ra
```

`# End of function`

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra

**Caller Saves Registers on Stack**

```c
void somefunc() {
    ...
    /*uses t0, t1 somewhere */
    ...
    t2 = small_func(t0);
    ...
}
```

```assembly
addi $sp, $sp, -12
sw $ra, 8($sp)
sw $t0, 4($sp)
sw $t1, 0($sp)
add $a0, $t0, $0
jal small_func
lw $t1, 0($sp)
lw $t0, 4($sp)
lw $ra, 8($sp)
addi $sp, $sp, 12
...
jr $ra
```

caller-saved: $t0-$t9, $a0-$a9, $v0-$v9. callee-saved: $s0-$s7, $ra
Recursive Factorial

1 factorial:
2    bgtz $a0, doit             # Argument > 0
3    li       $v0, 1                       # Base case, 0! = 1
4    jr       $ra                           # Return
5  doit:
6    addi   $sp, sp, -8              # Allocate stack frame
7    sw     $s0,($sp)                 # Position for argument n
8    sw     $ra,4($sp)                # Remember return address
9    move $s0, $a0              # Push argument
10   addi   $a0, a0, -1               # Pass n-1
11   jal      factorial                    # Figure v0 = (n-1)!
12   mul    $v0,$s0,$v0             # Now multiply by n, v0 = n*(n-1)!
13   lw      $s0,($sp)                  # Restore registers from stack
14   lw      $ra,4($sp)                 # Get return address
15   addi   $sp, sp, 8                  # Pop
16   jr      $ra                               # Return