MIPS Calling Convention (for CS64)

CS 64: Computer Organization and Design Logic
Lecture #10

Ziad Matni
Dept. of Computer Science, UCSB
Administrative

• I’m giving you more time to do Lab 5
  – New due date is Friday 2/17

• This means I’m also moving the due dates for the remaining labs out

• Syllabus schedule has been updated online
# Administrative

<table>
<thead>
<tr>
<th>Wk</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Mon 02/13 Lab</td>
<td>Tue 02/14 Lecture 10</td>
<td>Wed 02/15 Lecture 11</td>
<td>Thu 02/16</td>
<td>Fri 02/17 Lab 5 DUE</td>
</tr>
<tr>
<td>7</td>
<td>Mon 02/20 UCSB HOLIDAY</td>
<td>Tue 02/21 Lecture 12</td>
<td>Wed 02/22 Lecture 13</td>
<td>Thu 02/23</td>
<td>Fri 02/24 Lab 6 DUE</td>
</tr>
<tr>
<td>8</td>
<td>Mon 02/27 Lab Lab 7 DUE</td>
<td>Tue 02/28 Lecture 14</td>
<td>Wed 03/01</td>
<td>Thu 03/02 MIDTERM 2</td>
<td>Fri 03/03</td>
</tr>
<tr>
<td>9</td>
<td>Mon 03/06 Lab</td>
<td>Tue 03/07 Lecture 15</td>
<td>Wed 03/08 Lecture 16 Lab 8 DUE</td>
<td>Thu 03/09</td>
<td>Fri 03/10</td>
</tr>
<tr>
<td>10</td>
<td>Mon 03/13 Lab</td>
<td>Tue 03/14 Lecture 17</td>
<td>Wed 03/15 Lecture 18</td>
<td>Thu 03/16 Lab 9 DUE Lab 10 DUE</td>
<td>Fri 03/17</td>
</tr>
</tbody>
</table>
Lecture Outline

• MIPS Calling Convention

with multiple examples!

... and a fair amount of detail,
so pay attention!!!
STACK (LIFO) PUSH AND POP

- **Push** operations add elements to the stack, following the Last In First Out (LIFO) principle.
- **Pop** operations remove the top element from the stack.

Diagram illustrating the process:

1. Push 1
2. Push 2
3. Push 3
4. Push 4
5. Push 5
6. Push 6
7. Pop 6
8. Pop 5
9. Pop 4
10. Pop 3
11. Pop 2
12. Pop 1
Stack Push and Pop

- **PUSH** one or more registers
  - $sp -= 4 * (# of regs)
  - `sw $reg, N($sp)`

- **POP** one or more registers
  - `lw $reg, N($sp)`
  - $sp += 4 * (# of regs)
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

• **DEMO**: save_registers.asm
Please make sure you have read the MIPS Calling Convention for CS64 document found at:

http://www.cs.ucsb.edu/~zmatni/cs64w17/documentation/calling_convention/calling_convention.html

This is important material to understand
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

• Why do we care?
  – When you are asked by me to use this C.C. and you don’t, you will get no credit on the assignment or test question
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
  – 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
Instructions to Watch Out For

• `jal <label>` and `jar $ra` always go together

• Function arguments have to be stored ONLY in `$a0 thru $a3`

• Function return values have to be stored ONLY in `$v0 and $v1`

• If functions need additional registers whose values we don’t care about keeping after the call, then they can use `$t0 thru $t9`

• What about `$s` registers? AKA the `preserved registers`
  – Hang in there... will talk about them in a few slides...
MIPS C.C. for CS64: Assumptions

• Does 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

• Assumes that all values on the stack are always 32-bits
  – That is, no complex data structures like C-Structs, etc...
MIPS Call Stack

- We know what a Stack is...
- A “Call Stack” is used for storing **the return addresses** of the various **functions** which have been **called**

- When you **call** a function (e.g. `jal funcA`), the address that we need to return to is **pushed** into the call stack.

  ...

  *funcA* does it’s thing... then...

  ...

  The function needs to return.

So, the address is **popped** off the call stack.
void first()
{
    second();
    return;
}

void second()
{
    third();
    return;
}

void third()
{
    fourth();
    return;
}

void forth()
{
    return;
}

fourth:
    jr $ra

third:
    push $ra
    jal fourth
    pop $ra
    jr $ra

second:
    push $ra
    jal third
    pop $ra
    jr $ra

first:
    jal second
    li $v0, 10
    syscal
Why `addiu`? Because there is no such thing as a negative memory address. We want to avoid triggering a processor-level exception on overflow.

```assembly
fourth:
    jr $ra

third:
    addiu $sp, $sp, -4
    sw $ra, 0($sp)
    jal fourth
    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    jr $ra

second:
    addiu $sp, $sp, -4
    sw $ra, 0($sp)
    jal third
    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    jr $ra

first:
    jal second
li $v0, 10
syscall
```

```assembly
fourth:
    jr $ra

third:
    push $ra
    jal fourth
    pop $ra
    jr $ra

second:
    push $ra
    jal third
    pop $ra
    jr $ra

first:
    jal second
li $v0, 10
syscall
```
Functions that Call Functions That Need Additional Registers

- Consider this program:

- Say we use: $t0$ and $t1$ for vars $a$ and $b$ in `doSomething`

- Should we use $t2$ for sub in `subTwo`?

  ---NO!---

  (not according to the MIPS C.C. Rules since $t*$ are unpreserved regs)

- Suppose that we have more vars than just $a$ and $b$...
  - We’ll have a mess on our hands
Solution?
Preserved Registers in The Call Stack!

- See previous example: `save_registers.asm`
- Values in **unpreserved registers** (every register introduced so far except for `$sp`), **are not preserved across calls**.

- You must always assume that unpreserved registers **will change values across calls**, even if you know for a fact that they don't.

- **Why?**
  - Convention. Rules are Rules...
  - **Even if you have correct code**, not obeying these rules means that **you don’t have fully-independent functions**, which leads to **hacky code**.
int subTwo(int a, int b) {
    int sub = a - b;
    return sub;
}

int doSomething(int x, int y) {
    int a = subTwo(x, y);
    int b = subTwo(y, x);
    ...
    return a + b;
}
```c
int subTwo(int a, int b) {
    int sub = a - b;
    return sub;
}

int doSomething(int x, int y) {
    int a = subTwo(x, y);
    int b = subTwo(y, x);
    ...
    return a + b;
}
```
subTwo:
sub $t0, $a0, $a1
move $v0, $t0
jr $ra

doSOMething:
addiu $sp, $sp, -16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)
move $s0, $a0
move $s1, $a1
jal subTwo
lw $ra, 12($sp)
lw $s2, 8($sp)
lw $s1, 4($sp)
lw $s0, 0($sp)
addiu $sp, $sp, 16
jal $ra

int subTwo(int a, int b)
{
  int sub = a - b;
  return sub;
}

int doSOMething(int x, int y)
{
  int a = subTwo(x, y);
  int b = subTwo(y, x);
  ... return a + b;
}
Lessons Learned

• We passed arguments into the functions using $a*$

• We used $s*$ to work out calculations in registers that we wanted to preserve, so we made sure to save them in the call stack
  – These var values DO need to live beyond a call
  – In the end, the original values were returned back

• We used $t*$ to work out calcs. in regs that we did not need to preserve
  – These values DO NOT need to live beyond a function call

• We used $v*$ as regs. to return the value of the function
Invariants to Preserve

- **RULE:** The values that are held in preserved registers ($s0-$s7 and $sp) immediately before a function call must be the same immediately after the function returns.
  
  - While a call is occurring, the values in preserved registers are allowed to change, but the values must return to their original values after a call is performed

- **RULE:** The values that are held in unpreserved registers ($t0-$t9, $a0-$a3, and $v0-$v1) must always be assumed to change after a function call is performed.
  
  - After a function call is made, the programmer must assume that the values in unpreserved registers all changed in a completely unpredictable way
Another Example Using Recursion
Recursive Functions

• This same setup handles nested function calls and recursion
  – i.e. By saving $ra methodically on the stack

• Example: recursive_fibonacci.asm
Recall the Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, etc...

\[ fib(n) = fib(n - 1) + fib(n - 2) \]

In C/C++, we might write the recursive function as:

```c
int fib(int n) {
    if (n == 0) return (0);
    else if (n == 1) return (1);
    else return (fib(n-1) + fib(n-2));
}
```

**Base cases**
• We’ll need at least 3 registers to keep track of:
  – The (single) input to the call
  – The output (or partial output) to the call
  – The value of $ra (since this is a recursive function)

• If we make $s0 = n and $s1 = fib(n – 1)

• Then we need to save $s0, $s1 and $ra in mem
  – So that we do not corrupt/lose what’s already in these regs

• We’ll use $s0 registers b/c we need them beyond the function call
recursive_fibonacci.asm

• First: Check for the base cases
  – Is $n (a0) equal to 0 or 1?

• Next: 3 registers containing integers, means we need to plan for 3 words in the stack
  – **Push** 3 words in (i.e. 12 bytes)
  – $sp \rightarrow 12$
  – The order by which you put them in does *not strictly* matter, *but* it makes more “organized” sense to
    
    **push** $s0, s1, then $ra
recursive_fibonacci.asm

- **Next:** calculate fib(n – 1)
  - Call recursively, copy output in $s1
- **Next:** calculate fib(n – 2)
recursive_fibonacci.asm

- **Next:** calculate fib\(n - 1\)
  - Call recursively, copy output in \$s1
- **Next:** calculate fib\(n - 2\)
  - Call recursively, add output to \$s1
recursive_fibonacci.asm

- **Next**: calculate fib(n – 1)
  - Call recursively, copy output in $s1
- **Next**: calculate fib(n – 2)
  - Call recursively, add output to $s1
- **Next**: restore registers
  - Pop the 3 words back to $s0, $s1, and $ra
- **Next**: return to caller
  - Issue a `jr $ra` instruction

- Note how when we leave the function and go back to the “callee”, we did not disturb what was in the registers previously
- And now we have our output where it should be, in $v0
More Recursion! 😊

• Check out the demo file `tail_recursive_factorial.asm` at home.
• What’s special about the tail recursive functions (see example)?
  – Where the recursive call is the very last thing in the function.
  – With the right optimization, it can use a constant stack space.

```c
int recFac(int n, int accum) {
    if (n == 0) return accum;
    else return recFac(n - 1, n * accum);
}
```
Next Lesson

• Introduction to Logic Design!!!
YOUR TO-DOs

• Assignment #5: Due Fr. 2/17

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