CS61C Fall 2016 Discussion 4 – MIPS Procedures & CALL

1 MIPS Control Flow

2 Conventions

1. How should `$sp` be used? When do we add or subtract from `$sp`?

   `$sp` points to a location on the stack to load or store into. Subtract from `$sp` before storing, and add to `$sp` after restoring.

2. Which registers need to be saved or restored before using `jr` to return from a function?

   All `$s*` registers that were modified during the function must be restored to their value at the start of the function

3. Which registers need to be saved before using `jal`?

   `$ra`, and all `$t*`, `$a*`, and `$v*` registers if their values are needed later after the function call.

4. How do we pass arguments into functions?

   `$a0`, `$a1`, `$a2`, `$a3` are the four argument registers

5. What do we do if there are more than four arguments to a function?

   Use the stack to store additional arguments

6. How are values returned by functions?

   `$v0` and `$v1` are the return value registers.

When calling a function in MIPS, who needs to save the following registers to the stack? Answer "caller" for the procedure making a function call, "callee" for the function being called, or "N/A" for neither.

$0	$v*	$a*	$t*	$s*	$sp	$ra
N/A	Caller	Caller	Caller	Callee	N/A	Caller

Now assume a function `foo` calls another function `bar` (which may be called from a `main` fucntion), which is known to call some other functions. `foo` takes one argument and will modify and use `$t0` and `$s0`. `bar` takes two arguments, returns an integer, and uses `$t0-$t2` and `$s0-$s1`. In the boxes below, draw a possible ordering of the stack just before `bar` calls a function. The top left box is the address of `$sp` when `foo` is first called, and the stack goes downwards, continuing at each next column. Add '(f)' if the register is stored by `foo` and '(b)' if the register is stored by `bar`. The first one is written in for you.

1 `$ra` (f)	5 `$t0` (f)	9 `$v0` (b)	13 `$t1` (b)
2 `$s0` (f)	6 `$ra` (b)	10 `$a0` (b)	14 `$t2` (b)
3 `$v0` (f)	7 `$s0` (b)	11 `$a1` (b)	15
4 `$a0` (f)	8 `$s1` (b)	12 `$t0` (b)	16

3 A Guide to Writing Functions

4 C to MIPS

1. Assuming `$a0` and `$a1` hold integer pointers, swap the values they point to via the stack and return control.

```
void swap(int *a, int *b) {

    int tmp = *a;
    *a = *b;
    *b = tmp;

}
```

```
addiu $sp, $sp, -4
lw    $t0, 0($a0)
sw    $t0, 0($sp)
lw    $t0, 0($a1)
sw    $t0, 0($a0)
lw    $t0, 0($sp)
sw    $t0, 0($a1)
addiu $sp, $sp, 4
jr    $ra
```

2. Translate the following algorithm that finds the sum of the numbers from 0 to N to MIPS assembly. Assume `$s0` holds N, `$s1` holds `sum`, and that N is greater than or equal to 0.

```
int sum = 0

if (N==0)       return 0;

while (N != 0) {
    sum += N
    N--;
}

return sum;
```

```
Start: add $s1 $0 $0
Loop: beq   $s0, $0, Ret
add   $s1, $s1, $s0
addiu  $s0, $s0, -1
j     Loop
Ret: addiu  $v0, $0, 0
j     Done
Done: jr    $ra
```

3. What must be done to make the adding algorithm from the previous part into a callable MIPS function?

   Add a prologue and epilogue to reserve space on the stack and store all necessary variables (see #3). Use `$a0` instead of `$s0` to store N, the function's argument.

```
RecursiveSum:
addiu $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
li $v0, 0
beq $a0, $0, Ret
addiu $a0, $a0, -1
jal RecursiveSum
lw $a0, 0($sp)
addu $v0, $v0, $a0
Ret:
lw $ra, 4($sp)
addiu $sp, $sp, 8
jr $ra
```

5 Compile, Assemble, Link, Load, and Go!

5.1 Overview

5.2 Exercises

1. What is the Stored Program concept and what does it enable us to do?
   It is the idea that instructions are just the same as data, and we can treat them as such. This enables us to write programs that can manipulate other programs!

2. How many passes through the code does the Assembler have to make? Why?
   Two, one to find all the label adresses and another to convert all instructions while resolving any forward references using the collected label addresses.

3. What are the different parts of the object files output by the Assembler?
   Header: Size and position of other parts
   Text: The machine code
   Data: Binary representation of any data in the source file
   Relocation Table: Identifies lines of code that need to be "handled" by Linker
   Symbol Table: List of the files labels and data that can be referenced
   Debugging Information: Additional information for debuggers

4. Which step in CALL resolves relative addressing? Absolute addressing? Assembler, Linker.

5. What step in CALL may make use of the `$at` register? Assemble

6. What does RISC stand for? How is this related to pseudoinstructions?
   Reduced Instruction Set Computing. Minimal set of instructions leads to many lines of code. Pseudoinstructions are more complex instructions intended to make assembly programming easier for the coder. These are converted to TAL by the assembler.