CS61C Spring 2015 Discussion 4 – MIPS Procedures & CALL

1 MIPS Control Flow

There are only two instructions necessary for creating and calling functions: `jal` and `jr`. If you follow register conventions when calling functions, you will be able to write much simpler and cleaner MIPS code.

2 Conventions

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

$sp should be used to jump around machine language locally from the current line. We add/subtractfrom $sp to change the add

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

$ra- whoops actually it is just the $s* registers that need to be restored to their previous values.

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

$ra, $s*, whoops, also a* and v* registers if their values are needed later after the call.

4. How do we pass arguments into functions?

By saving them into a* registers

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

Hmm maybe store them on the stack and load them when needed? Or maybe use additional registers (t*? s*?).

Nope just use the stack.

6. How are values returned by functions?

Using the v* arguments.

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.

Ok, so the caller has to save everything, except the $s* which are the responsibility of the callee to save if it is goingto over write them

$0	$v*	$a*	$t*	$s*	$sp	$ra
n/a	caller	caller	n/a nope: caller	caller:nope callee	caller - nope: n/a. Actually, i think I disagree, it should be c	caller

Now assume a function `foo` (which may be called from a `main` fucntion) calls another function `bar`, 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 $ra (b)	9 $s0 (b)	13 $t2 (b)
2 $s0 (f)	6 $a0 (b)	10 $s1 (b)	14
3 $a0 (f)	7 $a1 (b)	11 $t0 (b)	15
4 $t0 (f)	8 $v0 (b)	12 $t1 (b)	16

hmmmm don't see where we need $v0 (f) that is in the answer key...

3 A Guide to Writing Functions

```
FunctionFoo:  # PROLOGUE
# begin by reserving space on the stack
addiu $sp, $sp, -FrameSize

# now, store needed registers
sw $ra, 0($sp)
sw $s0, 4($sp)
...
# BODY
...
# EPILOGUE
# restore registers
lw $s0 4($sp)
lw $ra 0($sp)

# release stack spaces
addiu $sp, $sp, FrameSize

# return to normal execution
jr $ra
```

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;

}
```

```
lw $t1, 0($a1)
sw $t1, 0($a0)
sw $t0, 0($a1)
j $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;
```

```
        addiu $s1, $zero, 0
        bne $s0, $zero, Loop
        addiu $v0, $zero, 0
        j Return

        add $s1, $s1, $s0
        addi $s1, $s1, -1
Loop:   bne $s1, $zero, Loop

        add $v0, $s0, $zero
        j $ra

Return
```

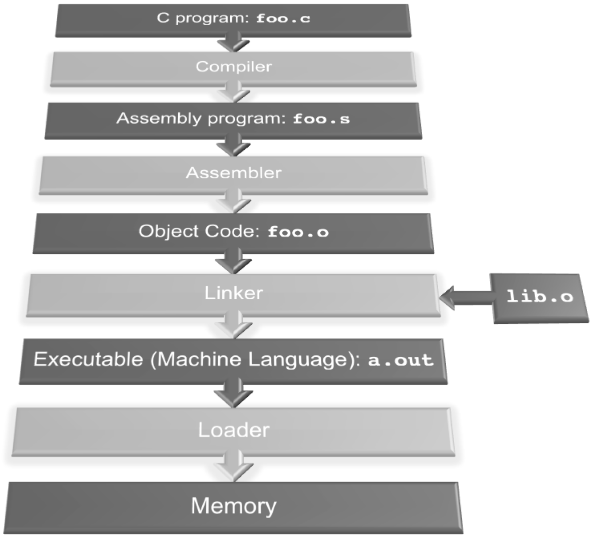
```
Ret:
lw $a0, 0($sp)
addu $v0, $v0, $a0
```

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

1) $ra, $v0, $s1, s0, need to be saved onto the stack (with space allocated) if necessary
2) the $sp should be reset when done
3) use $a0 for N since it is the argument

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 concept of placing information in memory and loading it to execute. It allows us to have much larger programs than register memory.

2. How many passes through the code does the Assembler have to make? Why?

   Twice; once to get the list of destinations and once to convert all instructions and put in the destinations

3. What are the different parts of the object files output by the Assembler?

   1) Header, size and location of other pieces
   2) text segment with machine code
   3) data in binary of the source code
   4) relocation info about what instructions must change if the program is moved
   5) symbol table linking addresses to names
   6) debugging info on how it was compiled

4. Which step in CALL resolves relative addressing? Absolute addressing?

   Assembly handles relative addressing while the linker handle absolute addressing

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
   Psuedo instructions can be builts from common combinations of base instructions