CS61C Discussion 3 – MIPS II/CALL

1 Translating between C and MIPS

Translate between the C and MIPS code. You may want to use the MIPS Green Sheet as a reference. We show you how the different variables map to registers – you don't have to worry about the stack or any memory-related issues.

C	MIPS

`// Nth_Fibonacci(n):`
`// $s0 -> n, $s1 -> fib`
`// $t0 -> i, $t1 -> j`
`// Assume fib, i, j are these values`
`int fib = 1, i = 1, j = 1;`
`if (n==0)      return 0;`
`else if (n==1) return 1;`
`n -= 2;`
`while (n != 0) {`
`    fib = i + j;`
`    j = i;`
`    i = fib;`
`    n--;`
`}`
`return fib;`	

2 MIPS Addressing

- We have several **addressing modes** to access memory (immediate not listed):
  a. **Base displacement addressing:** Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
  b. **PC-relative addressing:** Uses the PC (actually the current PC plus four) and adds the I-value of the instruction (multiplied by 4) to create an address (used by I-format branching instructions like beq, bne)
  c. **Pseudodirect addressing:** Uses the upper four bits of the PC and concatenates a 26-bit value from the instruction (with implicit 00 lowest bits) to make a 32-bit address (used by J-format instructions)
  d. **Register Addressing:** Uses the value in a register as a memory address (jr)

1. You need to jump to an instruction that $2^{28} + 4$ bytes higher than the current PC. How do you do it? Assume you know the exact destination address at compile time. (Hint: you need multiple instructions)

2. You now need to branch to an instruction $2^{17} + 4$ bytes higher than the current PC, when \$t0 equals 0. Assume that we're not jumping to a new 2^{28} byte block. Write MIPS to do this.

3. Given the following MIPS code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your green sheet!):

```
0x002cff00: loop: addu $t0, $t0, $t0      | 0 |    |    |    |    |       |
0x002cff04:       jal  foo                | 3 |                          |
0x002cff08:       bne  $t0, $zero, loop   | 5 | 8 |    |                 |
...
0x00300004: foo:  jr $ra                 $ra = _______________
```

3 MIPS Calling Conventions

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

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

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

4. How do we pass arguments into functions?

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

6. How are values returned by functions?

4 Writing MIPS Functions

Here is a general template for writing functions in MIPS:

```
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
```

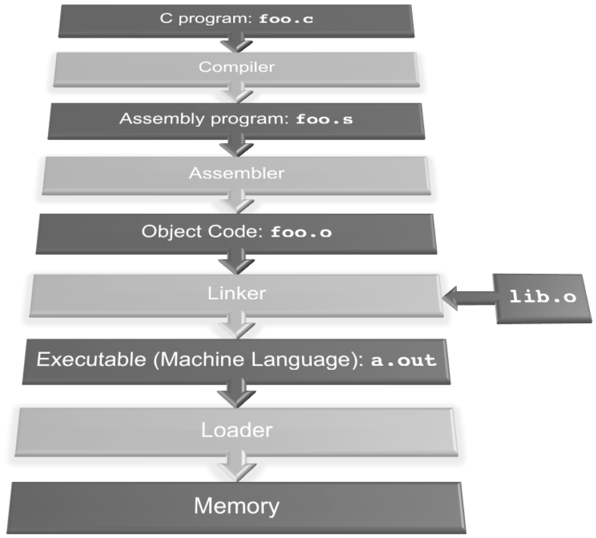
Translate the following C code for a recursive function into a callable MIPS function.

```
// Finds the sum of numbers 0 to N
int sum_numbers(int N) {
    int sum = 0

    if (N==0) {
        return 0;
    } else {
        return N + sum_numbers(N - 1);
    }
}
```

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?

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

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

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

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

6. What does RISC stand for? How is this related to pseudoinstructions?