CS61C Homework 3 - C to MIPS Practice Problems

TA: Sagar Karandikar

Spring 2015

This homework is an ungraded assignment designed to familiarize you with translating C code to MIPS. We will release solutions on Sunday, Feb 22nd, so that you may use them to study for the exam.

Problem 1 - Useful Snippets

In this section, we'll take the same problem (that of printing a string) and approach it using different C constructs. This should allow you to see how various C constructs are translated into MIPS.

Suppose that we have a `print` function, but that this function only takes one character and prints it to the screen. It expects the character to be in the lower 8 bits of $a0.

A) Translate into MIPS, while preserving the while loop structure:

```
void string_print(char *print_me) {
    while (*print_me != '\0') {
        print(*print_me);
        print_me++;
    }
}
```

Solution:

```
stringprint:
    # prologue, backup ra, backup s0
    addiu $sp, $sp, -8
    sw $ra, 0($sp)
    sw $s0, 4($sp)

    addiu $s0, $a0, 0 # copy a0 to s0 so we don't have to back it up
    lbu $a0, 0($s0) # load character for first iteration

Loop:
    beq $a0, $0, Ret # break out of loop if loaded character is null terminator
    jal print # call print (this is why we loaded to a0)
```

```
    addiu $s0, $s0, 1 # increment pointer
    lbu $a0, 0($s0) # load next character
    j Loop

Ret:
    # epilogue, restore ra, restore s0
    lw $s0, 4($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 8
    jr $ra
```

B) Translate into MIPS, while preserving the for loop structure (your function is given the string length):

```
void string_print(char *print_me, int slen) {
    for (int i = 0; i < slen; i++) {
        print(*(print_me+i));
    }
}
```

Solution:

```
stringprint:
    # prologue, backup ra, backup s0, s1, s2
    addiu $sp, $sp, -16
    sw $ra, 0($sp)
    sw $s0, 4($sp)
    sw $s1, 8($sp)
    sw $s2, 12($sp)

    addiu $s0, $a0, 0 # copy a0 to s0 so we don't have to back it up
    addiu $s1, $a1, 0 # copy a1 to s1 so we don't have to back it up
    addiu $s2, $0, 0 # initialize loop counter

Loop:
    beq $s2, $s1, Ret # break out of loop if i == slen
    addu $a0, $s2, $s0
    lbu $a0, 0($a0) # get first char
    jal print # call print (this is why we loaded to a0)
    addiu $s2, $s2, 1 # increment loop var
    j Loop

Ret:
```

```
    # epilogue, restore ra, restore s0, s1, s2
    lw $s2, 12($sp)
    lw $s1, 8($sp)
    lw $s0, 4($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 16
    jr $ra
```

C) Translate into MIPS, while preserving the do while loop structure:

```c
void string_print(char *print_me) {
    if (!(*print_me)) {
        return;
    }
    do {
        print(*print_me);
        print_me++;
    } while (*print_me);
}
```

Solution:

```
stringprint:
    # prologue, backup ra, backup s0
    addiu $sp, $sp, -8
    sw $ra, 0($sp)
    sw $s0, 4($sp)

    addiu $s0, $a0, 0 # copy a0 to s0 so we don't have to back it up
    lbu $a0, 0($s0) # load character for first iteration
    beq $a0, $0, Ret # do nothing if loaded character is null terminator

Loop:
    jal print # call print (this is why we loaded to a0)
    addiu $s0, $s0, 1 # increment pointer
    lbu $a0, 0($s0) # load next character
    bne $a0, $0, Loop # continue loop if loaded character is not null terminator

Ret:
    # epilogue, restore ra, restore s0
    lw $s0, 4($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 8
```

```
    jr $ra
```

Problem 2 - Recursive Fibonacci

Convert the following recursive implementation of Fibonacci to MIPS. Do not convert it to an iterative solution.

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

Solution:

```
# note: this solution is instructional and not necessarily optimized

fib:
    addiu $sp, $sp, -12
    sw $ra, 0($sp) #backup ra for recursive calls
    sw $a0, 4($sp) #backup a0 for recursive calls
    sw $s0, 8($sp) #backup s0 since we use it

    beq $a0, $0, ReturnZero
    addiu $t0, $0, 0
    slti $t0, $a0, 2 # you can also beq with one
    bne $t0, $0, ReturnOne

    addiu $a0, $a0, -1
    jal fib
    move $s0, $v0
    lw $a0, 4($sp)
    addiu $a0, $a0, -2
    jal fib
    add $v0, $v0, $s0

    lw $s0, 8($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 12
```

```
    jr $ra

ReturnZero:
    # epilogue just to make our returns consistent
    lw $s0, 8($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 12
    li $v0, 0
    jr $ra

ReturnOne:
    # epilogue just to make our returns consistent
    lw $s0, 8($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 12
    li $v0, 1
    jr $ra
```

Problem 3 - Memoized Fibonacci

Now, modify your recursive Fibonacci implementation to memoize results. For the sake of simplicity, you can assume that the array given to you (memolist) is at least n elements long for any n. Additionally, the array is initialized to all zeros.

```c
int fib(int n, int* memolist) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }
    if (memolist[n]) {
        return memolist[n];
    }
    memolist[n] = fib(n-1, memolist) + fib(n-2, memolist);
    return memolist[n];
}
```

Solution:

```
# note: this solution is instructional and not necessarily optimized

fib:
```

```
    addiu $sp, $sp, -12
    sw $ra, 0($sp) #backup ra for recursive calls
    sw $a0, 4($sp) #backup a0 for recursive calls
    sw $s0, 8($sp) #backup s0 since we use it

    beq $a0, $0, ReturnZero
    addiu $t0, $0, 0
    slti $t0, $a0, 2 # you can also beq with one
    bne $t0, $0, ReturnOne

    # not the n = 1 or n = 0 cases, check memoized table
    sll $t0, $a0, 2 # convert n to offset (ints are 4 bytes)
    addiu $t0, $t0, $a1 # add offset to base pointer ($a1)
    lw $v0, 0($t0)
    bne $v0, $0, RetMemo

    # not in table, compute it the old-fashioned way
    addiu $a0, $a0, -1
    jal fib
    move $s0, $v0
    lw $a0, 4($sp)
    addiu $a0, $a0, -2
    jal fib
    add $v0, $v0, $s0

    # store copy in the memoize table
    lw $a0, 4($sp) # first, restore a0
    sll $t0, $a0, 2 # convert n to offset (ints are 4 bytes)
    addiu $t0, $t0, $a1 # add offset to base pointer ($a1)
    sw $v0, 0($t0)

    # epilogue
    lw $s0, 8($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 12
    jr $ra

ReturnZero:
    # epilogue just to make our returns consistent
    lw $s0, 8($sp)
    lw $ra, 0($sp)
```

```
    addiu $sp, $sp, 12
    li $v0, 0
    jr $ra

ReturnOne:
    # epilogue just to make our returns consistent
    lw $s0, 8($sp)
    lw $ra, 0($sp)
    addiu $sp, $sp, 12
    li $v0, 1
    jr $ra

RetMemo: # returns value already in v0
    # epilogue
    lw $s0, 8($sp)
    lw $ra, 0($s0)
    addiu $sp, $sp, 12
    jr $ra
```

Problem 4 - Self-Modifying MIPS

Write a MIPS function that performs identically to this code when called many times in a row, but does not store the static variable in the static segment (or even the heap or stack):

```
short nextshort() {
    static short a = 0;
    return a++;
}
```

Tips/Hints:

- You can assume that the `short` type is 16 bits wide. `short`s represent signed numbers.
- You can assume that your MIPS code is allowed to modify any part of memory.
- See the title of this question.

Solution:

```
nextshort:
    addiu $v0, $0, 0
    la $t0, nextshort
    lw $t1, 0($t0)
    addiu $t3, $0, 0xFFFF
```

```
    and $t2, $t1, $t3
    beq $t2, $t3, HandleSpecial
    addiu $t1, $t1, 1
    sw $t1, 0($t0) # store incremented instruction
    jr $ra # ret value is already in v0
HandleSpecial: # here, handle the overflow case
    la $t6, backupinst
    lw $t1, 0($t6)
    sw $t1, 0($t0)
    jr $ra # ret value is already in v0

backupinst:
    addiu $v0, $0, 0
```