CS 241 - Week 3 Tutorial Solutions

Assembly Language Programming and C++ Review

Fall 2018

1 Assembly Language Programming

Problem 1 - I/O and loops in MIPS

Adapt your solution to problem 4 from last tutorial to read in characters from stdin until EOF is encountered and print out their uppercase versions to stdout. If the character is not a lower-case letter, simply print out the character unchanged.

Solution:

```
;$27 is stdin, $28 is stdout
lis $27
.word 0xffff0004
lis $28
.word 0xffff000c

;$25 = 'a', the first lowercase letter.
;Characters less than $25 are not lowercase.
lis $25
.word 97

;$26 = 'z', the last lowercase letter.
;Characters more than $26 are not lowercase.
lis $26
.word 122

;$20 is 'A' - 'a', the amount that
;we should add to make a character uppercase
lis $20
.word -32

;$24 is EOF
lis $24
```

```
        .word -1

        ;Load characters until EOF is encountered
loop:   lw $3, 0($27)
        beq $3, $24, end

        ;If $3 < $25, we can print this unchanged
        slt $5, $3, $25
        bne $5, $0, print

        ;If $26 < $3, we can print this unchanged
        slt $5, $26, $3
        bne $5, $0, print

        ;We are lowercase, add $20
        add $3, $3, $20

print:  sw $3, 0($28)
        beq $0, $0, loop

end:    jr $31
```

Problem 2 - Using the stack in MIPS

Write a MIPS program which reads in characters from stdin until EOF is encountered, then prints the same characters out backwards to stdout. Use the stack to store the characters.

Solution:

```
        ;$27 is stdin, $28 is stdout
        lis $27
        .word 0xffff0004
        lis $28
        .word 0xffff000c

        ;$24 is EOF
        lis $24
        .word -1

        ;$4 is 4
        lis $4
        .word 4

        ;$26 is the initial value of $30
        add $26, $30, $0

        ;Load characters until EOF is encountered
loop:   lw $3, 0($27)
        beq $3, $24, end

        ;Push the character
        sw $3, -4($30)
        sub $30, $30, $4

        ;Repeat
        beq $0, $0, loop

end:

        ;Pop characters until $30 is back where it started
loop2:  beq $26, $30, end2

        ;Pop a character
        add $30, $30, $4
        lw $3, -4($30)
```

```
        ;Print the character
        sw $3, 0($28)

        ;Repeat
        beq $0, $0, loop2

end2:   jr $31
```

Problem 3 - Functions and recursion in MIPS

Rewrite your solution to Problem 1 from last tutorial using a recursive function instead of a loop.

Solution:

```
;The first instance of fact is implicitly called by the start of the program

fact:   sw $31,  -4($30)
        sw $1,   -8($30)
        sw $11, -12($30)
        lis $31
        .word 12
        sub $30, $30, $31

        lis $11
        .word 1

        ;If $1 = 0, base case of $3 = 1
        bne $1, $0, recur
        add $3, $11, $0
        beq $0, $0, clean

        ;Call fact with $1 - 1
recur:  sub $1, $1, $11
        lis $31
        .word fact
        jalr $31

        ;Restore value of $1
        add $1, $1, $11

        ;Multiply previous answer by $1 to get new factorial
        mult $3, $1
        mflo $3

clean:  lis $31
        .word 12
        add $30, $30, $31

        lw $11, -12($30)
        lw $1,   -8($30)
        lw $31,  -4($30)
        jr $31
```

2 C++ Review

2.1 Code Style

Here are some problems about the code and possible ways we could improve it

1. Inconsistent Spacing. Indentation and use of spaces should be consistent throughout the program.

2. Repeated use of long type names. Mistake can be easily made when typing long, specific sequence of characters over and over again. Furthermore, these code are invulnerable when implementation changes are required. When there are complicated type, we can use typedef.

3. Redundant if statement. For example, in **foo**, the entire function could be replaced by simply returning the result of comparison.

4. Pass large data structures by value is inefficient, since the entire data structure are copied once the function is called. Pass by reference are often preferred.

5. Some code quick be efficiently solved and nicely presented by using the correct data structures. Make proper use of STL as much as possible. e.g. **baz** could be done by using **std::set**.

6. Magic numbers should be avoided entirely. All constant value should have a purpose, and a descriptive name.

7. Use **enum** constants to create a range of name constants. e.g. in **baz**, all string literals could be defined as enum.

8. Naming for functions and variables should be more descriptive.

9. Documentations are always the essential part of good style!