CS 21 Lab 4

2019-10009

2)

a) (slt $at, $t1, $t2) then (bne $at, $0, some_label) This is correct as it breaks down the blt pseudoinstruction into two parts, first doing the comparison and storing it in the $at register then the next instruction is a basic branching instruction that checks if it really if $t1 is really less than $t2 by comparing the output of the first instruction to 0.

b) (slt $at, $t2, $t1) then (beq $at, $0, some_label) First we check if $t2 is less than $t1. Then check if $t2 is less than $t1 in our branching instruction, if not it means $t2 is greater than or equal to $t1 (complement) and it also implies $t1 is less than or equal to $t2.

c) (slt $at, $t2, $t1) then (bne $at, $0, some_label) First we check if $t2 is less than $t1. Then check if $t2 is less than $t1 in our branching instruction, if it is it implies that $t1 is greater than $t2.

d) (slt $at, $t1, $t2) then (beq $at, $0, some_label) First we check if $t1 is less than $t2. Then check if $t1 is less than $t2 in our branching instruction, if it is not, it means that $t1 should be greater than or equal to $t2 by logical complement.

e) (bgez $0, some_label)

This is equivalent to the b pseudo instruction as this instruction will always be true since $0 will always be equal to zero. Hence, we branch to some_label.

3)

a) Line by line explanation in the frommips.c file

b) strspn

c) First, the strings Arr1 and Arr2 will now be integer arrays containing the values that are hard coded in our mips code. The hardcoded values of

CONST_ARR1_LEN and CONST_ARR2_LEN should also be changed accordingly. Note that we still don't include the CONST_ELEM_SIZE because pointer arithmetic adapts to the type that we have.

5)

a)

```
li $t0, 1
sw $t0, 0($sp)
lw $t0, 0($sp)
bge $t0, $t1, L3
lw $t0, 0($sp)
sw $t0, 4($sp)
lw $t0, 0($sp)
addiu $t0, $t0, 1
sw $t0, 0($sp)
```

b) The logical and operation is correctly expressed int the mips code as the branching instructions are consecutively placed. The mips code checks for the complement of j > 0 which is if j ≤ 0 (blez $t0, L2), it branches out of the loop once that condition is met. For A[j-1] > A[j], they first get accessed by some pointer arithmetic, then they are stored into $t3 and $t4 respectively. After, they are compared in the line "ble $t3, $t4, L2". Which branches out it $t3 (A[j-1]) is less then $t4 (A[j]). Hence, it works perfectly. Note that j is stored and loaded from the stack at 4($sp).

c)

```
xor $t3, $t3, $t4
xor $t4, $t3, $t4
xor $t3, $t3, $t4
```

This technique works for all values of A[j] and A[j-1] because of the boolean algebra behind this. Let x = $t3 and y =$t4,
1: x = x xor y,
2: y = x xor y, y = x xor y xor y
y = y xor y xor x (through commutativity)

y = x

3: x = x xor y, x = x xor y xor x xor y xor y

x = x xor x xor y xor y xor y (commutativity)

x = y