2.6

```
sll  $t0, $t3, 9    # shift $t3 left by 9, store in $t0
srl  $t0, $t0, 15   # shift $t0 right by 15
```

Problem 2. Bubble Sort

```
            addi $t2, $s1, -1   # $t2 is i

startiloop: addi $t7, $zero, 1  # $t7 is j
            addi $t0, $s0, 0    # $t0 is pointer to the j-1th array elt
            addi $t1, $s0, 4    # $t1 is pointer to the jth array elt

startjloop: lw $t3, 0($t0)
            lw $t4, 0($t1)
            slt $t5, $t1, $t0
            beq $t5, $zero, endjloop
            sw $t3, 0($t1)
            sw $t4, 0($t0)

 endjloop:  addi $t0, $t0, 4
            addi $t1, $t1, 4
            addi $t7, $t7, 1
            slt $t5, $t2, $t7
            beq $t5, $zero, startjloop

            addi $t2, $t2, -1
            slt $t5, $t2, $zero
            beq $t5, $zero, startiloop
```

2.29

```
        add  $t0, $zero, $zero    # initialize running sum $t0 = 0
loop:   beq  $a1, $zero, finish   # finished when $a1 is 0
        add  $t0, $t0, $a0        # compute running sum of $a0
        sub  $a1, $a1, 1          # compute this $a1 times
        j    loop
finish: addi $t0, $t0, 100        # add 100 to a * b
        add  $v0, $t0, $zero      # return a * b + 100
```

The program computes a * b + 100.

2.37

Pseudoinstruction	What it accomplishes	Solution
`move $t1, $t2`	$t1 = $t2	`add $t1, $t2, $zero`
`clear $t0`	$t0 = 0	`add $t0, $zero, $zero`
`beq $t1, small, L`	if ($t1 == small) go to L	`li $at, small` `beq $t1, $at, L`
`beq $t2, big, L`	if ($t2 == big) go to L	`li $at, big` `beq $at, $zero, L`
`li $t1, small`	$t1 = small	`addi $t1, $zero, small`
`li $t2, big`	$t2 = big	`lui $t2, upper(big)` `ori $t2, $t2, lower(big)`
`ble $t3, $t5, L`	if ($t3 <= $t5) go to L	`slt $at, $t5, $t3` `beq $at, $zero, L`
`bgt $t4, $t5, L`	if ($t4 > $t5) go to L	`slt $at, $t5, $t4` `bne $at, $zero, L`
`bge $t5, $t3, L`	if ($t5 >= $t3) go to L	`slt $at, $t5, $t3` `beq $at, $zero, L`
`addi $t0, $t2, big`	$t0 = $t2 + big	`li $at, big` `add $t0, $t2, $at`

`lw $t5, big($t2)`	$t5 = Memory[$t2 + big]	`li $at, big`
		`add $at, $at, $t2`
		`lw $t5, $t2, $at`

Note: In the solutions, we make use of the `li` instruction, which should be implemented as shown in rows 5 and 6.

3.9 The problem is that `A_lower` will be sign-extended and then added to `$t0`. The solution is to adjust `A_upper` by adding 1 to it if the most significant bit of `A_lower` is a 1. As an example, consider 6-bit two's complement and the address 23 = 010111. If we split it up, we notice that `A_lower` is 111 and will be sign-extended to 111111 = –1 during the arithmetic calculation. `A_upper_adjusted` = 011000 = 24 (we added 1 to 010 and the lower bits are all 0s). The calculation is then 24 + –1 = 23.

3.10 Either the instruction sequence

```
addu $t2, $t3, $t4
sltu $t2, $t2, $t4
```

or

```
addu $t2, $t3, $t4
sltu $t2, $t2, $t3
```

works.

3.12 To detect whether `$s0` < `$s1`, it's tempting to subtract them and look at the sign of the result. This idea is problematic, because if the subtraction results in an overflow, an exception would occur! To overcome this, there are two possible methods: You can subtract them as unsigned numbers (which never produces an exception) and then check to see whether overflow would have occurred. This method is acceptable, but it is lengthy and does more work than necessary. An alternative would be to check signs. Overflow can occur if `$s0` and (`-$s1`) share the same sign; that is, if `$s0` and `$s1` differ in sign. But in that case, we don't need to subtract them since the negative one is obviously the smaller! The solution in pseudocode would be

```
if ($s0<0) and ($s1>0) then
      $t0:=1
else if ($s0>0) and ($s1<0) then
      $t0:=0
else
      $t1:=$s0-$s1
      if ($t1<0) then
            $t0:=1
      else
            $t0:=0
```