Question 5 (30 points): In this question you will write `convertCommandsToString`, a subroutine that receives two parameters:

`$a0`: address of a word that contains the binary representation for commands

`$a1`: address of `stringCommandBuffer`, a buffer of characters where the string commands should be written

There are 16 binary commands in the binary representation of the commands. Each individual command is represented in two consecutive bits. The first command is formed by the two most-significant bits (bits 31-30) and the last command is formed by the two least-significant bits (bits 1-0). This is the correspondence between binary commands and string commands:

Binary	String
00	left
01	right
10	up
11	down

The subroutine `convertCommandsToString` must create a string representation of the commands that starts with the character '`<`', is followed by the string representation of the commands corresponding to the binary commands, separated by comma, and ends with the character '`>`'. For instance, if the binary representation of the commands is:

```
010011 ... 1000
```

Then `convertCommandsToString` will create the string:

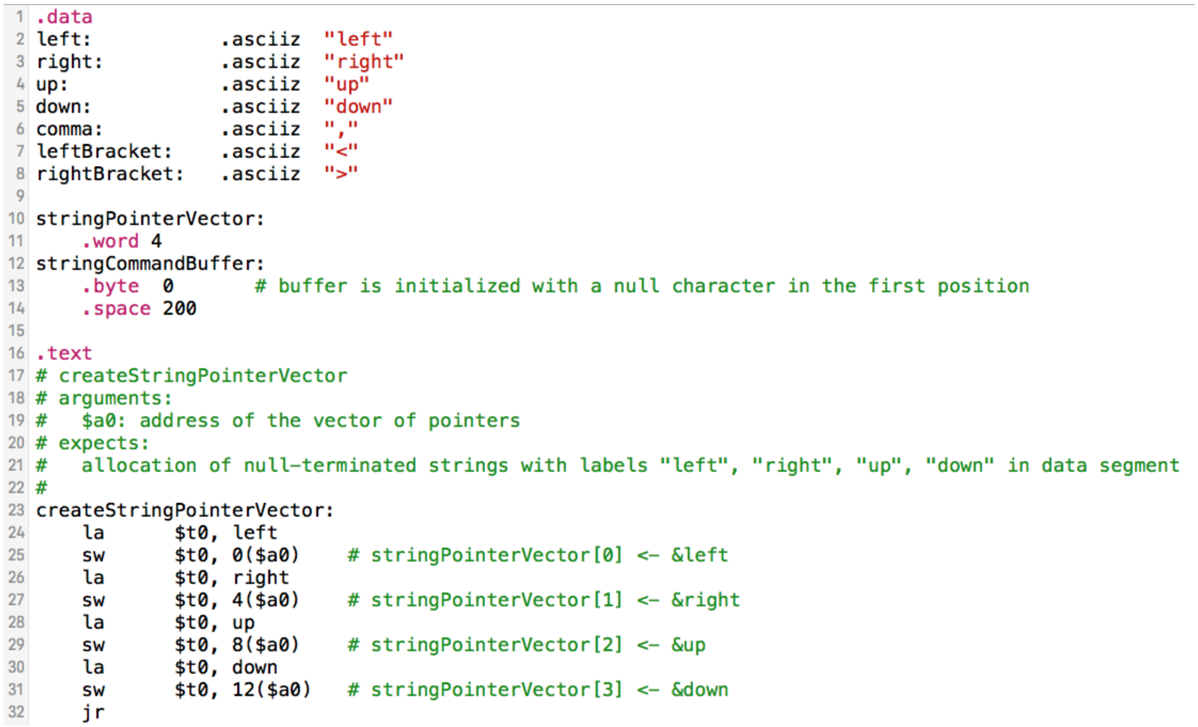
```
<right,left,down, ... ,up,left>
```

Where `...` in the example above represents other commands that appear in the binary representation and in the string representation.

The person that is passing this task to you already started the organization of the data in memory and created the strings that you need in the data segment. She also has written a subroutine that creates a vector of pointers to the strings `left`, `right`, `up`, and `down`. You should expect that the function `createStringPointerVector` will be invoked before `convertCommandsToString` with the following invocation:

```
la      $a0, stringPointerVector
jal     createStringPointerVector
```

Thus you can be sure that at the address of `StringPointerVector` your subroutine will find a vector of pointers to strings as created by `createStringPointerVector`. You should use this vector when implementing `convertCommandsToString`. You should also invoke `concatenate` to append strings to the buffer when implementing `convertCommandsToString` — You can use `concatenate` in this



```
.data
left:           .asciiz  "left"
right:          .asciiz  "right"
up:             .asciiz  "up"
down:           .asciiz  "down"
comma:          .asciiz  ","
leftBracket:    .asciiz  "<"
rightBracket:   .asciiz  ">"

stringPointerVector:
    .word 4
stringCommandBuffer:
    .byte  0       # buffer is initialized with a null character in the first position
    .space 200

.text
# createStringPointerVector
# arguments:
#   $a0: address of the vector of pointers
# expects:
#   allocation of null-terminated strings with labels "left", "right", "up", "down" in data segment
#
createStringPointerVector:
    la      $t0, left
    sw      $t0, 0($a0)    # stringPointerVector[0] <- &left
    la      $t0, right
    sw      $t0, 4($a0)    # stringPointerVector[1] <- &right
    la      $t0, up
    sw      $t0, 8($a0)    # stringPointerVector[2] <- &up
    la      $t0, down
    sw      $t0, 12($a0)   # stringPointerVector[3] <- &down
    jr
```

Figure 1: Data Segment and subroutine `createStringPointerVector`

question even if you did not write a correct solution to the previous question. You must follow all the subroutine invocation conventions of MIPS.

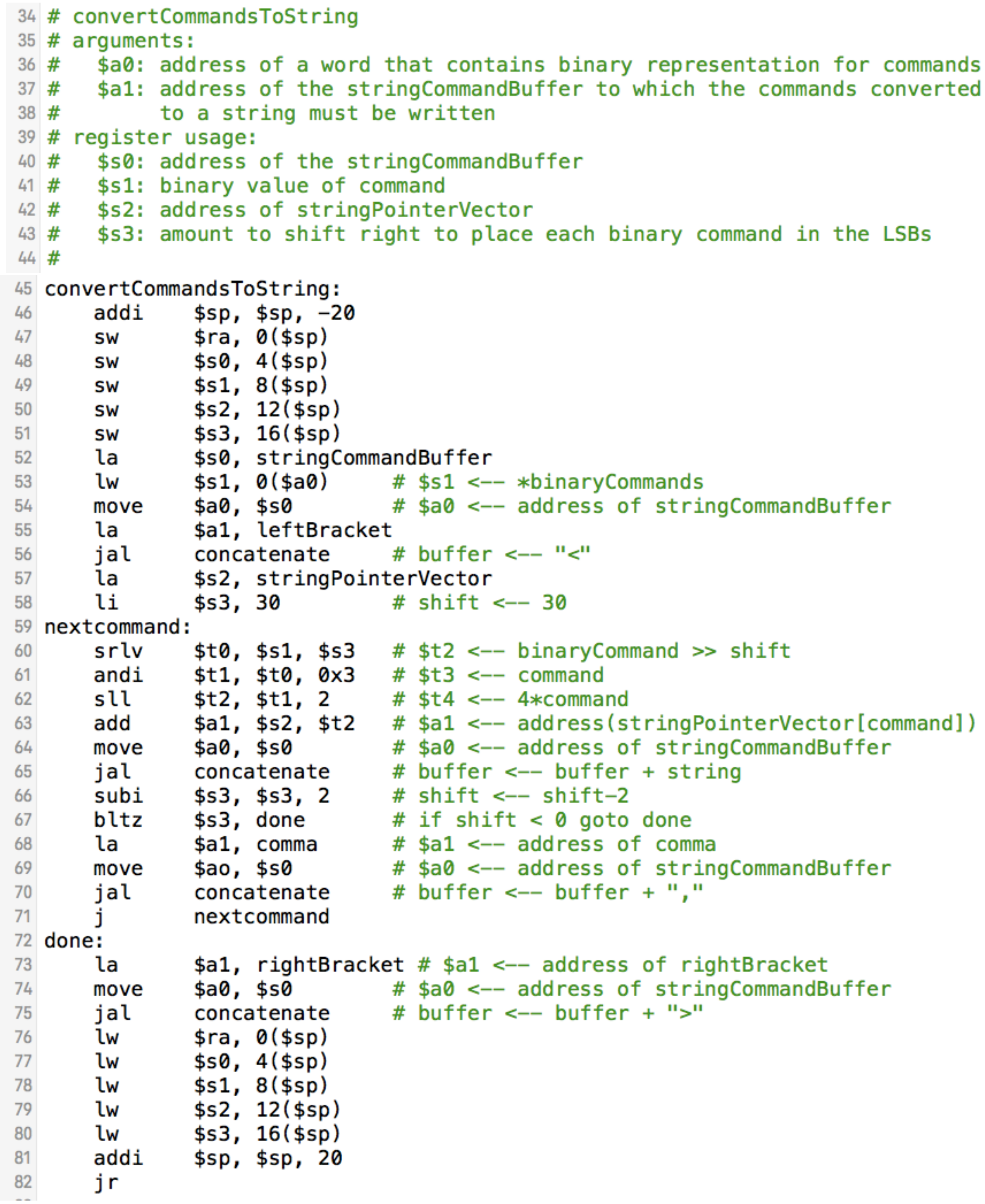
The data segment and the code for `createStringPointerVector` are shown in Figure 1.

The solution presented in Figure 2 is illustrated in Figure 3. Figure 3(a) shows the state of the memory and the values in registers `$a0` and `$a1` when the `convertCommandsToString` is first invoked. Notice that the provided code for `createStringPointerVector` has already been executed and thus the vector of pointers `stringPointerVector` has already been created. Notice also that the space for the `stringCommandBuffer` has already been allocated and that the character zero (equivalent to the sentinel value \0) has been put in the first byte of this buffer effectively creating a null-terminated string to which we can concatenate.

Figure 3(b) illustrates the processing of the first binary command. Observe that the value of the binary command is first loaded into `$s1` before the value of `$a0` is changed to the address of the first character in the `stringCommandBuffer`. Then the first binary command is isolated through an `srlv` instruction — the number of bits that will be shifted will change for each iteration of the loop so that each time we isolate a different binary command.

After the `srlv` instruction, it is necessary to mask only the two Least Significant Bits (LSBs) with an `andi` instruction because in later iterations of the loop non-zero bits belonging to other digital commands will appear in the upper bits of that word.

Now the value of the command can be used to index the `stringPointerVector`. To do that we have to multiply the value of the command by four (using a `sll` instruction) and then add to the base



```
# convertCommandsToString
# arguments:
#   $a0: address of a word that contains binary representation for commands
#   $a1: address of the stringCommandBuffer to which the commands converted
#        to a string must be written
# register usage:
#   $s0: address of the stringCommandBuffer
#   $s1: binary value of command
#   $s2: address of stringPointerVector
#   $s3: amount to shift right to place each binary command in the LSBs
#
convertCommandsToString:
    addi    $sp, $sp, -20
    sw      $ra, 0($sp)
    sw      $s0, 4($sp)
    sw      $s1, 8($sp)
    sw      $s2, 12($sp)
    sw      $s3, 16($sp)
    la      $s0, stringCommandBuffer
    lw      $s1, 0($a0)      # $s1 <-- *binaryCommands
    move    $a0, $s0         # $a0 <-- address of stringCommandBuffer
    la      $a1, leftBracket
    jal     concatenate      # buffer <-- "<"
    la      $s2, stringPointerVector
    li      $s3, 30          # shift <-- 30
nextcommand:
    srlv    $t0, $s1, $s3    # $t2 <-- binaryCommand >> shift
    andi    $t1, $t0, 0x3    # $t3 <-- command
    sll     $t2, $t1, 2      # $t4 <-- 4*command
    add     $a1, $s2, $t2    # $a1 <-- address(stringPointerVector[command])
    move    $a0, $s0         # $a0 <-- address of stringCommandBuffer
    jal     concatenate      # buffer <-- buffer + string
    subi    $s3, $s3, 2      # shift <-- shift-2
    bltz    $s3, done        # if shift < 0 goto done
    la      $a1, comma       # $a1 <-- address of comma
    move    $ao, $s0         # $a0 <-- address of stringCommandBuffer
    jal     concatenate      # buffer <-- buffer + ","
    j       nextcommand
done:
    la      $a1, rightBracket # $a1 <-- address of rightBracket
    move    $a0, $s0         # $a0 <-- address of stringCommandBuffer
    jal     concatenate      # buffer <-- buffer + ">"
    lw      $ra, 0($sp)
    lw      $s0, 4($sp)
    lw      $s1, 8($sp)
    lw      $s2, 12($sp)
    lw      $s3, 16($sp)
    addi    $sp, $sp, 20
    jr
```

Figure 2: A solution to the `convertCommandsToString` subroutine.

of the array (which is currently in `$s0`). The figure shows the content of the `stringCommandBuffer` after both the string `'<'` and the string `'right'` have both been contatenated.

Figure 3(b) illustrated the execution of the next iteration of the look when the next command

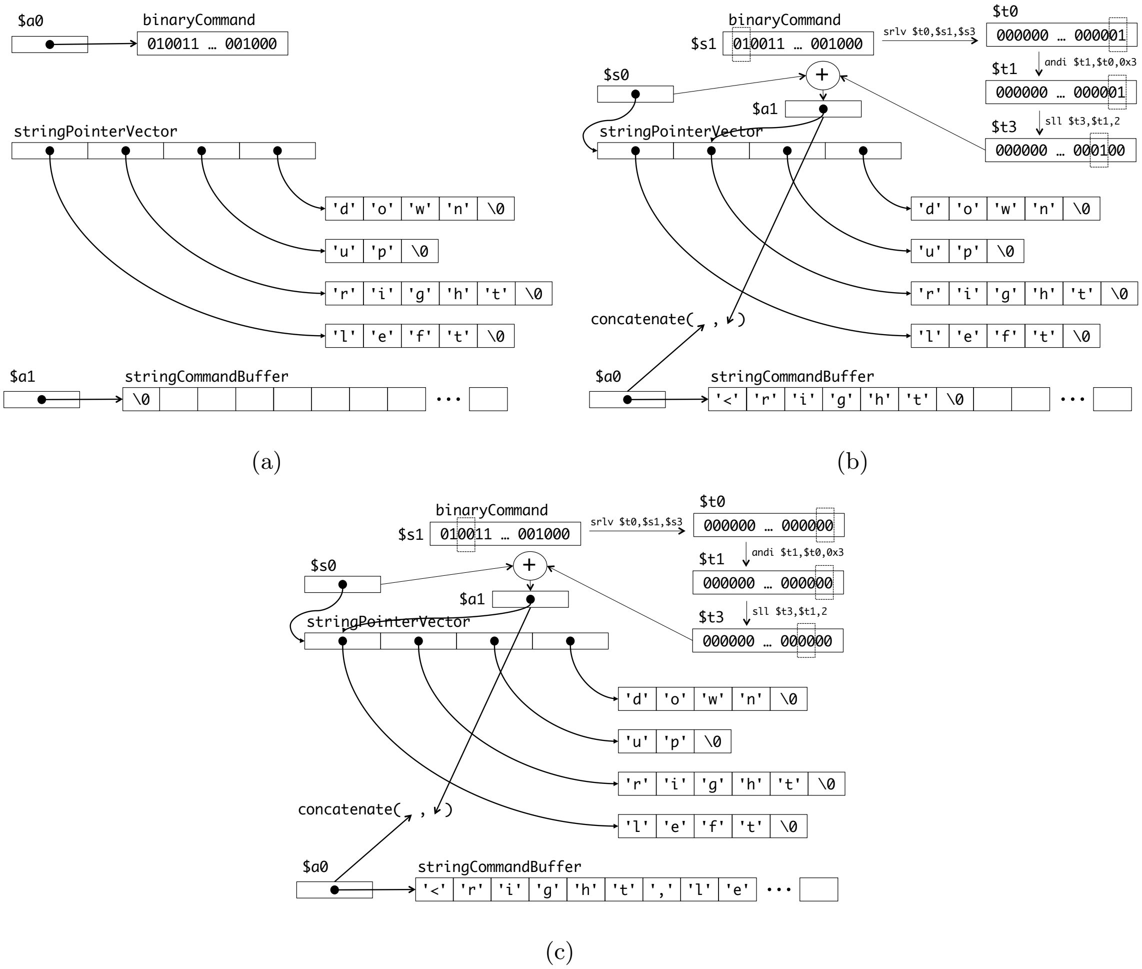


Figure 3: Illustration of the solution for `convertCommandsToString`.

— 00 in the example — is processed.