►Solution◄

Question 1: (25 points)

Consider the MIPS assembly code of the function `mysteryProc` given below. Assume that this implementation adheres to the MIPS procedure-calling conventions. Also, note that only function-local variables are stored in $\$s_x$ registers. For simplicity, the MIPS code for storing and restoring callee-saved registers to and from the stack are omitted.

```
(1)  0x0040 0000    mysteryProc: addi  $t1, $zero, 32
(2)  0x0040 0004                 sllv  $s0, $s0, $t1
(3)  0x0040 0008             L1: add   $t2, $a0, $zero
(4)  0x0040 000C                 lbu   $t3, 0($t2)
(5)  0x0040 0010                 bne   $t3, $zero, L2
(6)  0x0040 0014                 j     L3
(7)  0x0040 0018             L2: addi  $a0, $a0, 1
(8)  0x0040 001C                 addi  $s0, $s0, 1
(9)  0x0040 0020                 j     L1
(10) 0x0040 0024             L3: add   $v0, $zero, $s0
(11) 0x0040 0028                 jr    $ra
```

a. (5 points) How many parameters does the function `mysteryProc` have? Give a name for each parameter. You will use these names in your source code for part c of this question. Also, indicate the type of each parameter, i.e., whether it is an address or a value. Justify your answer.

> **Solution:** The function accesses register `$a0`. Therefore, it has one parameter. The value in `$a0` is added to `$zero` at line 3, and the result (`$t2`) is then used as the base-address for a load. Thus, `$a0` is probably the base of an array, and thus is a pointer. Also, note that we are loading a byte using the base address `$a0` at line 4, and also incrementing the base address by 1 at line 7. Therefore, `$a0` is probably the base address of a character array. We will call this array `str`.

b. (10 points) The MIPS implementation of `mysteryProc` given above is intentionally naïve and is not the best written code. Optimize this code to implement the same functionality but using as few and/or higher-performing MIPS instructions as possible.

> **Solution:**
> Here's the first optimized version, where we use `li` to initialize `$t0 = 0` and remove the first two `addi` and `sllv` instructions. Then, we replace the `bne` instruction with `beq`, which allows us to get rid of the `j L3` instruction.
>
> ```
> (1)  mysteryProc: li    $s0, 0
> ```

```
(2)           L1: lbu   $t1, 0($a0)
(3)               beq   $t1, $zero, L2
(4)               addi  $a0, $a0, 1
(5)               addi  $s0, $t0, 1
(6)               j     L1
(7)           L2: add   $v0, $zero, $s0
(8)               jr    $ra
```

We can further optimize the above code to remove the `j L1` instruction from the loop `L1`. Here's the resulting code:

```
(1)  mysteryProc: li    $s0, 0
(2)               lbu   $t1, 0($a0)
(3)               beq   $t1, $zero, L2
(4)           L1: addi  $a0, $a0, 1
(5)               addi  $s0, $s0, 1
                  lbu   $t1, $0($a0)
(6)               bne   $t1, $zero, L1
(7)           L2: add   $v0, $zero, $s0
(8)               jr    $ra
```

c. (10 points) In class, we looked at several examples of mapping C-style high-level code to MIPS assembly code. In this question, your task is to reverse engineer the assembly code of the function `mysteryProc` to provide a C-style code that best represents the given code. The code is printed with line numbers to facilitate referencing to instructions in your answer. Note that you may find it easier to reverse engineer your optimized code rather than the given code. In a single sentence, write down what this function does.

Solution: Here are the steps to reverse-engineer the given MIPS assembly code:

- Following the discussion in a, we know that the function parameter is `char* str`.
- Also note that, in the second-last line of the given/optimized code, the function initializes the return-value register `$v0`. Therefore, `mysteryProc` must return a value. The type of this return value is probably a signed `int`, because at line 10, `$v0` is assigned the value of `$s0`, which is initialized to 0 in line 2 and later incremented by 1 inside the loop L2 at line 8.
- Note that only function-local variables are stored in $\$s_x$ registers. Therefore, $\$s0$ must be a local variable. We will call it `x`.
- Next, we will comment the assembly code.

```
(1) mysteryProc: li   $s0, 0          # x = 0
(2)              lbu  $t1, 0($a0)     # t1 = str[0]
(3)              beq  $t1, $zero, L2  # if (str[0] == '\0' goto L2)
(4)          L1: addi $a0, $a0, 1     # $a0++ i.e., increment index of str
(5)              addi $s0, $s0, 1     # x++
                 lbu  $t1, $0($a0)    # t1 = str[1]
(6)              bne  $t1, $zero, L1  # if (str[0] != '\0' loopback to L1);
(7)          L2: add  $v0, $zero, $s0 # v0 = x
(8)              jr   $ra             # return(v0)
```

Initially, `$a0` contains the base address of `str`, and is incremented to index each element of the character array `str`. Putting together all the information from above observations and comments, here is the equivalent C-style code:

```
int mysteryProc (char* str) {
    int x = 0;
    while (*str != '\0') {
        str++;
        x++;
    }
    return x;
}
```

The `mysteryProc` function takes as an argument a pointer to a null-terminated string and returns its size.