CS224
Section No.: 01
Spring 2018
Lab No.: 01
Deniz Yüksel / 21600880

Solution of 1:

```
        .text
.globl __start
__start:
        # Print out the request.
        li $v0, 4
        la $a0, prompt
        syscall
        # Get the number of elements the user will want in the array.
        li $v0, 5
        syscall
        # Store the number of elements inside $t0.
        move $t0, $v0
        # Also in $t4 and $t5, store the number of elements for later use.
        addi $t4, $t0, 0
        addi $t5, $t0, 0
        # Load the array to $t1.
        la $t1, array
        li $v0, 4
        la $a0, askForInt
        syscall
        # Loop for getting all the elements to the array.
Loop1:
        li $v0, 5
        syscall
        move $t2, $v0
        sw $t2, 0($t1)
        addi $t0, $t0, -1
        addi $t1, $t1, 4
        bgt $t0, $zero, Loop1

        mul $t4, $t4, 4  # RECOVER $t4.
        sub $t1, $t1, $t4           # $t1 = $t1 - 4*$t4
        li $v0, 4
        la $a0, printListMsg
        syscall
Loop2:
        li $v0, 1
```

```
        lw $a0, 0($t1)
        syscall
        addi $t1, $t1, 4
        addi $t4, $t4, -4
        bgt $t4, $zero, Loop2
        li $v0, 4
        la $a0, printReverseMsg
        syscall
        addi $t1, $t1, -4 # There is a bug in $t1. It points to 0 after Loop2.
ReversePrint:
        li $v0, 1
        lw $a0, 0($t1)
        syscall
        addi $t1, $t1, -4
        addi $t5, $t5, -1
        bgt $t5, $zero, ReversePrint
        .data
array:          .space 80
prompt:         .asciiz " Please enter the number of elements you want in the array. Maximum is 20:
"
askForInt:      .asciiz " \n Enter your integers. Each time, press enter: "
printListMsg:   .asciiz " \n Printing your list..."
printReverseMsg:.asciiz " \n Now printing in REVERSE! "
```

Solution of 2:

```
        .text
        .globl __start

__start:

        # Store ( c - d).
        lw $t1, c
        lw $t2, d
        # Finding absolute value of ( c - d)...

        bgt $t1, $t2, cGreater
        blt $t1, $t2, cLesser

cGreater:
        sub $t3, $t1, $t2
        sw $t3, y       # Store $t3 in y.
        li $v0, 4
        la $a0, cBigger
```

```
        syscall

        # Print the value of c - d.
        li $v0, 1
        lw $a0, y
        syscall

        jal SubtractTwoUntil
        # NOW FINALLY FINDING MOD 2

cLesser:
        sub $t3, $t2, $t1
        sw $t3, y
        li $v0, 4
        la $a0, cSmaller
        syscall

        # Print the value of d - c.
        li $v0, 4
        la $a0, absoluteValue
        syscall
        li $v0, 1
        lw $a0, y
        syscall

        jal SubtractTwoUntil
        # NOW FINALLY FINDING MOD 2

SubtractTwoUntil:
        addi $t3, $t3, -2
        bgt $t3, 1, SubtractTwoUntil
        sw $t3, x          # Store $t3 in x.

        li $v0, 4
        la $a0, finalValue
        syscall
        li $v0, 1
        lw $a0, x
        syscall

        beq $t3, 1, OddResult
        beq $t3, 0, EvenResult

OddResult:
        li $v0, 4
```

```
        la $a0, oddReport
        syscall
        li $v0, 10
        syscall

EvenResult:
        li $v0, 4
        la $a0, evenReport
        syscall
        li $v0, 10
        syscall

        .data
c: .word -25
d: .word 8
x: .word 0
y: .word 0 #absolute value
oddReport: .asciiz " \n | c - d| is an odd number. ( c - d) % 2 is 1. "
evenReport: .asciiz " \n | c - d|  is an even number. ( c - d) % 2 is 0. "
cBigger: .asciiz " \n C is greater than D "
cSmaller: .asciiz " \n C is smaller than D "
absoluteValue: .asciiz " \n Absolute value is "
finalValue: .asciiz " \n Final value is \n "
```

Solution of 3:

```
        la      $t1, a
        la      $t2, b
        .....
        .data
str:    .asciiz "\nHello\n"
a:      .word   1, 2, 3, 4
b:      .word   1
```

When load address instruction is called, the assembler calls two I type of instructions. For I type of instructions, we have 6 bits of opcode, 5 bits of rs, 5 bits of rt and 16 bits of immediate value. However, not all I type instructions are called directly. la is one of them. In order for assembler to call load address, first it calls load upper immediate with $1 and $00001001_{hex}$. And then calls or immediate  with the new value in $1 which is the first value inside the program, which is now $00001001_{hex}$:
Opcode, rs, rt, imm ( respectively): 001111  00000 00001 0001 0000 0000 0001
Now we group four by four which will be the object code of lui: 3c011001.

The next instruction is or immediate. The address will be loaded in $t1 so the register is $9 as we know from the green card. The assembler ors it with $00000008_{hex}$ with the value $00001001_{hex}$:
Opcode, rs, rt, imm ( respectively): 001101  00001  01001  0000 0000 0000 0100
We again group four by four: 34290008.
Now a is loaded in $t1.

Second statement is loading the address of b to $t2. This is again load address, so it calls load upper immediate and or immediate afterwards.
Opcode, rs, rt, imm ( respectively): 001111 00000 00001 0001 0000 0000 0000 000
Now, we group four by four: 3c011001.

Fort the next instruction, we will now use $t2 instead of $t1 in the previous load address instruction.
Opcode, rs, rt, imm ( respectively): 001101 0001 01010 0000 0000 0001 0100
Then we group four by four to obtain the object code in hex: 342a0018.

Solution of 4:

a. Symbolic machine instruction:  Symbolic machine instructions are the abbreviations of instructions that humans can understand. For example, “addi” is a symbolic machine instruction. This is it’s mnemonic name. But we can understand its real, longer name which is add immediate. Symbolic machine instructions are assembled by the assembler and are converted to machine instructions. Like Java or C++, Assembly can be called an integrated development environment for machine instructions.

b. Machine instruction: Machine instructions are operation code which are depicted in hexadecimal form. These instructions are executed in CPU. When symbolic machine instructions are called, they are translated to machine instructions that machines can understand. This code that machines can understand is called the object code.

c. Assembler directive: Assembler directives are the labels which start with “.” in MIPS. When they are detected by the assembler, the program will understand what do to in those segments. For example, below .data part we declare our data members. Under .text part, the instructions are written. Assembler directives are shown with pink color as default in MARS.

d. Pseudo instruction: Pseudo instructions are the instructions that humans can understand, which contain branch less than (blt), branch greater than (bgt), branch less than or equal (ble), branch greater than or equal (bge), load immediate (li) and move (move). This information is taken from MIPS Green Card.