```
###########################################################
#           cs315 Week 2 - part 2
#
#   -> MIPS code structure, clock cycle, improve code efficiency
#
###########################################################

Structure on MIPS program:

    ###########################################################
    #       Lab 2           <-- title of the file
    #   Name: Amir          <-- name of author
    #   Date: 1/1/2016      <-- date that program has been written
    #
    #   Description:        <-- brief description of the program
    #
    #       This program prints Hello World on to the screen.
    #       It then takes input and then prints the input on to the screen.
    #
    ###########################################################
    #       Register Usage  <-- register usage section. Shows the name of registers being use and their their purpose
    #   $t0 First number
    #   $t1 Second number
    ###########################################################
            .data           <-- .data directive. Statical variables should be initialized in .data section

    hello_p:    .asciiz "Hello World!\n"        <-- hello_p variables has a ASCII content of "Hello World" with end of line character (.asciiz)
    num_p:      .asciiz "Enter integer: "                                                                                      ^
    total_p:    .asciiz "The Integer is: "                                                                    pay attention to 'z'
    ###########################################################
            .text           <-- .text directive. Actual program or instructions will be placed in this section
    main:                   <-- 'main' label. It shows start of the program. Program without main does not run

    ** system call '4' which is print string. This block of code prints 'hello_p' variable
        li $v0, 4           # print hello world
        la $a0, hello_p
        syscall

    ** system call '4' which is print string. This block of code prints 'num_p' variable
        li $v0, 4           # prompt for and read number
        la $a0, num_p
        syscall

    ** system call '5' which is read integer. This block of code reads integer and returns the result in register $v0
        li $v0, 5
        syscall

    ** copies the content that is in register $v0 into register $t0
        move $t0, $v0       # move input to temp register

    ** system call '4' which is print string. This block of code prints 'total_p' variable
        li $v0, 4           # display label for the number. (i.e. Integer is:)
        la $a0, total_p
        syscall

    ** system call '1' which is print integer. The integer that is going to be printed should be placed in register $a0
```

```
    li $v0, 1           # print the number
    move $a0, $t0
    syscall
```

** system call '10' which is halt or terminate program. Program throws an exception is does not terminate. Do not forget this block.

```
    li $v0, 10          # end program
    syscall
###########################################################
```

Clock cycle:

To find the clock cycles:

1) Look at appendix D and check is we have any macro instruction. If we a have macro instruction, then use the last column to find clock cycle.
2) If we do not have a macro instruction, then use appendix A to find the time complexity or clock cycle of the instruction.
3) add the clock cycles for each instruction to find the total clock cycles.

Ex. lets find the clock cycles of the following code:

```
    li $v0, 1           --> 1 clock cycle
    li $a0, -100        --> 2 clock cycles
    syscall             --> 1 clock cycle

    li $t2, 16          --> 1 clock cycles
    lw $t1, 0($t7)      --> 1 clock cycle
    mul $t1, $t1, $t2   --> 33 clock cycles
    sw $t1, 0($t7)      --> 1 clock cycle
---------------------------------------------
    1 + 2 + 1 + 1 + 1 + 33 + 1 = 40  clock cycles
```

Now, lets rewrite the code to make it significantly faster. Lets look at instruction that takes most clock cycles (i.e. mul):

as we can see, we multiplied content of register $t1 by 16 (or 2^4)

```
    $t1 <-- memory[$t7]
    $t1 <-- $t1 * 16
    memory[$t7] <-- $t1
```

we can use property of shifting to reduce the clock cycles of multiplication:

shift right once is the same as division by 2

```
    Ex: 100 => 4    before shift right
        010 => 2    after shift right
```

shift left once is the same as multiplication by 2

```
    Ex: 010 => 2    before shift left
        100 => 4    after shift left
```

Therefore, multiplication by 16 is the same as shift left by 4. Now lets replace multiplication with shift left instruction:

```
    li $v0, 1           --> 1 clock cycle
    li $a0, -100        --> 2 clock cycles
    syscall             --> 1 clock cycle

    li $t2, 16          --> 1 clock cycles
    lw $t1, 0($t7)      --> 1 clock cycle
    sll $t1, $t1, 4     --> 1 clock cycles
    sw $t1, 0($t7)      --> 1 clock cycle
---------------------------------------------
```

1 + 2 + 1 + 1 + 1 + 1 + 1 = 8 clock cycles

44 vs. 8 clock cycles. Improvement.