# FIT1008: Introduction to Computer Science (FIT2085: for Engineers)
# Tutorial 3 Solutions
### Semester 1, 2019

## Exercise 1

If you were having trouble figuring out what the code does, don't be surprised. It contained the same mistake in lines 19 and 22, one that you do very often: trying to use `sw` to copy a register (**$t0** and **$t1**, respectively) into another register (**t1**). This is not what the sw instruction is for (it copies the content of a register into memory, not into another register), which is why the code does not even compile in MIPS.

But you should have been able to understand the code up to line 19, the fact there were mistakes, and what the general intent of the code was. It hopefully also created a lot of discussion at the tute. The following is a possible way to comment the (fixed) code:

```
1        .data
2  a:    .space 4
3  b:    .space 4
4        .text
5        # read integer and store it in a
6        addi $v0, $0, 5
7        syscall
8        sw $v0, a
9
10       # read integer and store it in b
11       addi $v0, $0, 5
12       syscall
13       sw $v0, b
14
15       # if a > b result = a else result = b
16       lw $t0, a              # $t0 = a
17       lw $t1, b              # $t1 = b
18       slt $t0, $t1, $t0      # if b < a then $t0 = 1
19       beq $t0, $0, one       # if $t0 = 0 (if not b < a) jump to one
20
21       lw $a0, a              # result = a,
22       j two                  # jump to end
23
24 one:  lw $a0, b              # result = b, could also have been add $a0, $t1, $0
25
26 two:  addi $v0, $0, 1        # print result
27       syscall
```

Now that the code is commented, it should be clearer it computes in **$t0** the maximum of the **a** and **b** and prints it. Note that line 19 in the original code cannot be fixed by copying **$t0** into **$a0** using something like add **$a0**, **$t0**, 0, since by the time line 19 is executed **$t0** no longer contains the value of variable **a**. This is a classical mistake that happens when trying to optimise the code, rather than constantly reloading the values from memory, which is why we ask you to be faithful in your own translations.

The code would be even clearer if, of course, it was correct, if the labels were more meaningful (say **else** and **endif**), and there was an initial lable saying **print_max** or something like that.

The associated Python code could have been something like:

```python
a = int(input())
b = int(input())
if b < a:
    result = a
else:
    result = b
print(result)
```

## Exercise 2

The Python code reads a number **n** and, while it is greater than 1, prints it and then either divides it by 2 (using integer division) if it is even, or multiplies it by 3 and adds 1, if it is odd. Once the loop is finished (the integer division results in 1, or the number is originally less than 1), it prints the resulting number.

A possible faithful translation of the Python code is as follows:

```
         .data
enter_integer_prompt: .asciiz   "Enter integer: "
n:       .word    0

         .text
         # Print "Enter integer: "
         la $a0, enter_integer_prompt      # load address of prompt into $a0
         addi $v0, $0, 4       # set syscall to 4
         syscall               # print prompt

         # Read integer and store into n
         addi $v0, $0, 5       # set syscall to 5
         syscall               # read int
         sw $v0, n             # n = the integer just read

while:   # while condition
         lw $t0, n             # $t0 = n
         addi $t1, $0, 1       # $t1 = 1
         slt $t2, $t1, $t0     # if 1 < n $t2 =1, else $t2 = 0
         beq $t2, $0, endwile    # if not 1 < n goto end of while loop

         # body of while n > 1

         # print n
         lw $a0, n             # load n into $a0
         addi $v0, $0, 1       # set syscall to 1
         syscall               # print n

         # if n%2 == 0
         lw $t1, n             # $t0 = n
         addi $t2, $0, 2       # $t2 = 2
         div $t1, $t2          # n / 2
         mfhi $t0              # $t0 = n%2 (reminder)
         bne $t0, $0, else     # if not n%2 == 0, goto else

         # body of then (n = n//2)
         lw $t1, n             # $t0 = n
         sra $t1, 1            # $t1 = n//2 (by shifting right arithmetic 1 bit)
         sw $t1, n             # n = n//2

         j endif               # Go to endif, jumping over the else part

else:    # we now know n%2 != 2
         # body of else (n = 3*n+1)
         lw $t1, n             # $t1 = n
         addi $t2, $0, 3       # $t2 = 3 for multiplication
         mult $t1, $t2         #  n * 3
         mflo $t1              # $t1 = 3*n
         addi $t1, $t1, 1      # $t1 = 3*n + 1
         sw $t1, n             # n = 3*n + 1
endif:
         j while               # Go back to while loop
endwhile:   # End while (print(n))
         lw $a0, n             # load n into $a0
         addi $v0, $0, 1       # set syscall to 1
         syscall               # print n

     # Terminate the program
         addi $v0, $0, 10      # set $v0 to 10
```

```
        syscall                 # terminate
```

Note that the loop condition not **n** > 1 could also have been done by testing **n** < 2. Not very faithful though. Also note that j **endif** could have been done this time as j **while** since there is nothing after the if-then-else within the while loop. Again, this is not very faithful. Finally, for those who know integer representation and understand bitwise arithmetic, the check for **n**%2 == %0 could also have been done as follows:

```
        # if n%2 == 0
        lw $t1, n               # $t0 = n
        andi $t2, $t1, 1        # $t2 = least bit of n (1 if n is odd, 0 if even)
        bne $t2, $0, else       # if not n%2 == 0, goto else
```

I was not expecting you to do this, but I do expect you to recognise what it does if I show it to you.

# Exercise 3

The code reads the size **z** of array **y**, creates the array, and then starts a loop that reads all its elements, storing in **x** the sum of all the elements read. Finally, if the array is non-empty, it prints **x**.

A possible faithful translation of the Python code is as follows:

```
         .data
prompt1: .asciiz "Enter integer: "
prompt2: .asciiz "Enter another integer: "
prompt3: .asciiz "Result: "

x:      .word  0 # x was initialised to 0 in Python
z:      .space 4 # no initial value provided in Pytho for z (is read), so we use .space
y:      .space 4 # y is a pointer to the array location; we do not yet know its address
i:      .space 4 # variable i was not initialised in Python either

        .text
        # Print "Enter integer: "
        la $a0, prompt1        # load address of prompt1 into $a0
        addi $v0, $0, 4        # set syscall to 4
        syscall                # print prompt

        # read z
        addi $v0, $0, 5
        syscall
        sw $v0, z

        # create a list of z elements
        addi $v0, $0, 9  # allocate space
        lw $t0, z
        sll $t1, $t0, 2  # z*4
        addi $a0, $t1, 4 # (z*4)+4
        syscall
        sw $v0, y        # y=address
        sw $t0, ($v0)    #y.length=z

        sw $0, i         # i=0 to start loop
 loop:  # while i < z
        lw $t0, i              # i
        lw $t1, z              # z
        slt $t0, $t0, $t1      # is i < z?
        beq $t0, $0, endloop   # if not i<z go to endloop

        # Print "Enter another integer: "
        la $a0, prompt2        # load address of prompt2 into $a0
        addi $v0, $0, 4        # set syscall to 4
        syscall                # print prompt
```

```
        # read y[i]
        lw $t0, i           # i
        lw $t1, y           # y
        sll $t0, $t0, 2     # i*4
        add $t0, $t0, $t1   # &(y+i*4)
        addi $v0, $0, 5     # read item
        syscall
        sw $v0, 4($t0)      # y[i]=item

        # x += y[i]
        lw $t0, x           # x
        lw $t1, i           # i
        lw $t2, y           # y
        sll $t1, $t1, 2     # i*4
        add $t2, $t2, $t1   # &(y+i*4)
        lw $t3, 4($t2)      # $t3 = y[i]
        addi $t0, $t0, $t3  # x + y[i]
        sw $t0, x           #x = x+ y[i]

        # i += 1
        lw $t0, i          #i
        addi $t0, $t0, 1   # i+1
        sw $t0, i          #i=i+1

        # restart the loop
        j loop

endloop:
        # Print "Result: "
        la $a0, prompt3      # load address of prompt3 into $a0
        addi $v0, $0, 4      # set syscall to 4
        syscall              # print prompt

        #print x
        lw $a0, x            #$a0 = x
        addi $v0, $0, 1      # print x
        syscall

        # Terminate the program
        addi $v0, $0, 10     # set $v0 to 10
        syscall              # terminate
```

The above would have been quite easy if you looked at the example I put on the lecture 6 slides (there is MIPS code there for reading a list with user-defined size). While using **.space** 4 is expected for the array and for **z**, it is OK if you decided to use **.word** 0 for **i**, as the compiker had to change the for into a while in any case.

# Exercise 4

Instructions sra and sll can be used to perform integer division and multiplication, respetively by a multiple of 2. This is because shifting a binary integer to the left by one bit while adding one zero on its right, is the same as multiplying that integer by 2. Consider for example the binary representation of decimal 2, which is 00010 (using 5 bits). If you shift that to the left by 1, you obtain 00100, which is 4 (that is 2*2). And If you shift again, you obtain 01000, which is 8 (that is, 2*4).

Similarly, shifting an integer one bit to the right while adding a copy of the MSB to the left, is the same as performing integer division by 2 on the number by 2. You can easily see that for positive numbers by going from the 01000 used above (8 in decimal), to 00100 shifting one to the right (4 in decimal), and to 00010 shifting another (2 in decimal). For negative numbers you don't really need to know. If you are curious, MIPS uses two's complement (as most architectures use). If you then take -4, which is 11100 and shift it 1 to the right copying the MSB you get 11110, which is -2 in decimal.

Therefore, a naive way to perform $6 \times 8$ in MIPS is as follows:

```
      .text
main: addi $t0, $0, 6
      addi $t1, $0, 8
      mult $t0, $t1
      mflo $t1
```

A better (more efficient) option is:

```
      .text
main: addi $t1, $0, 6
      sll  $t1, $t1, 3
```

# Exercise 5

A possible extension would have the following lines in the data segment:

```
is_palindrome_prompt:   .asciiz "The list is a palindrome\n"
not_palindrome_prompt:  .asciiz "The list is not a palindrome\n"
```

and the following lines in the text segment (note that since we do not have Python code for the extension, we do not need to be faithful... although as you can see, it makes it less clear):

```
        sw $0, i            # i=0 to start loop
        lw $t6, z           # $t6 will contain z during the lopp
        lw $t7, y           # $t7 will contain y during the loop
        sra $t8, $t1, 1     # $t8 will contain z//2 during the loop
        addi $t9, $t6, -1   # $t9 will contain z-1 during the loop

loop:   # while i < z//2
        lw $t0, i           # i
        slt $t0, $t0, $t8   # is i < z//2?
        beq $t0, $0, is_palindrome   # if not i<z//2 go to endloop, which means it is palindrome

        # $t2 = y[i]
        lw $t0, i           # i
        sll $t0, $t0, 2     # i*4
        add $t1, $t0, $t7   # &(y+i*4)
        lw $t2, 4($t1)      # $t2 = y[i]

        # $t3 = y[z-1-i]
        lw $t0, i           # i
        sub $t1, $t9, $t0   # z-1-i
        sll $t1, $t1, 2     # (z-1-i)*4
        add $t1, $t7, $t1   # &(y+(z-1-i)*4)
        lw $t3, 4($t1)      # $t3 = y[z-1-i]

        # if $t2 != $t3, it is not palindrome
        bne $t2, $t3, not_palindrome

        # i += 1
        lw $t0, i          #i
        addi $t0, $t0, 1   # i+1
        sw $t0, i          #i=i+1
```

```
        # restart the loop
        j loop

not_palindrome:
    #print "The list is not a palindrome"
    la $a0 not_palindrome_prompt
    li $v0 4
    syscall

        # Terminate the program
        addi $v0, $0, 10    # set $v0 to 10
        syscall             # terminate

is_palindrome: #end of the lopp, so it is palindrome
    #print "The list is a palindrome"
    la $a0 is_palindrome_prompt
    li $v0 4
    syscall

        # Terminate the program
        addi $v0, $0, 10    # set $v0 to 10
        syscall             # terminate
```