**Samuel Youssef**

**March 16, 2018**

**CSC 34200 - Computer Organization**

Instructor: Prof. Zheng Peng

Lab **03**

* **Task 1:**

1. The entire source code for task 1 is:

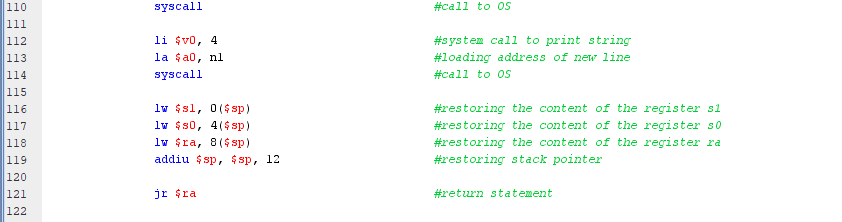
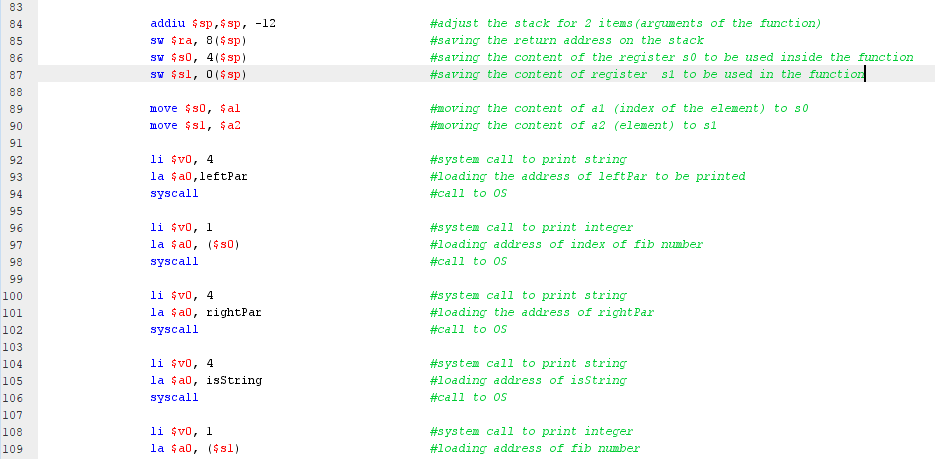
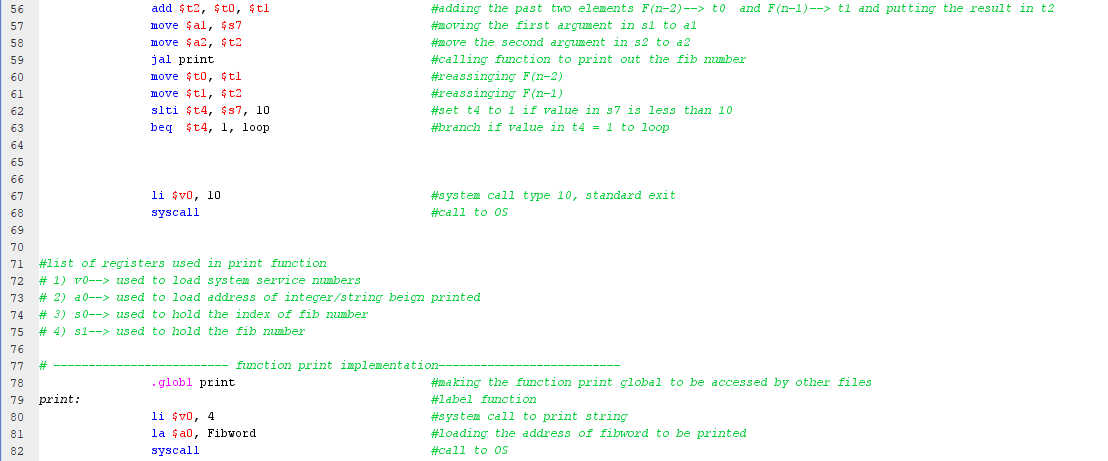
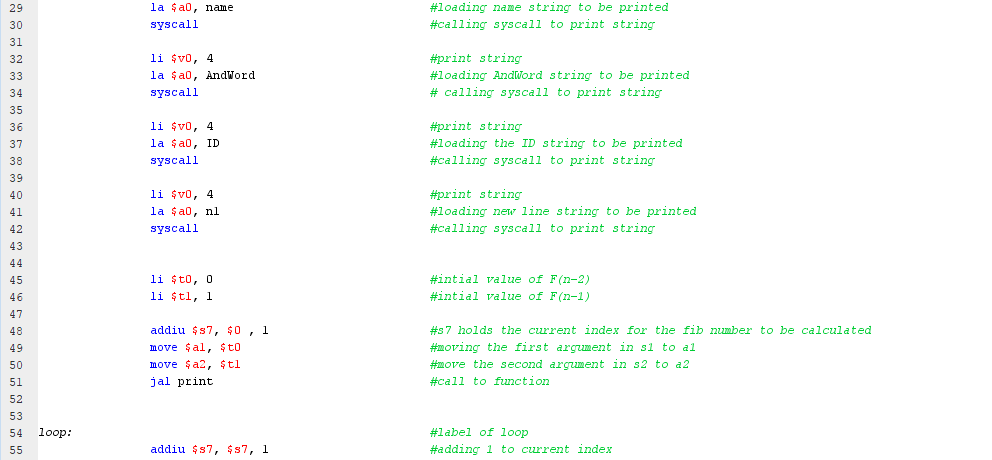
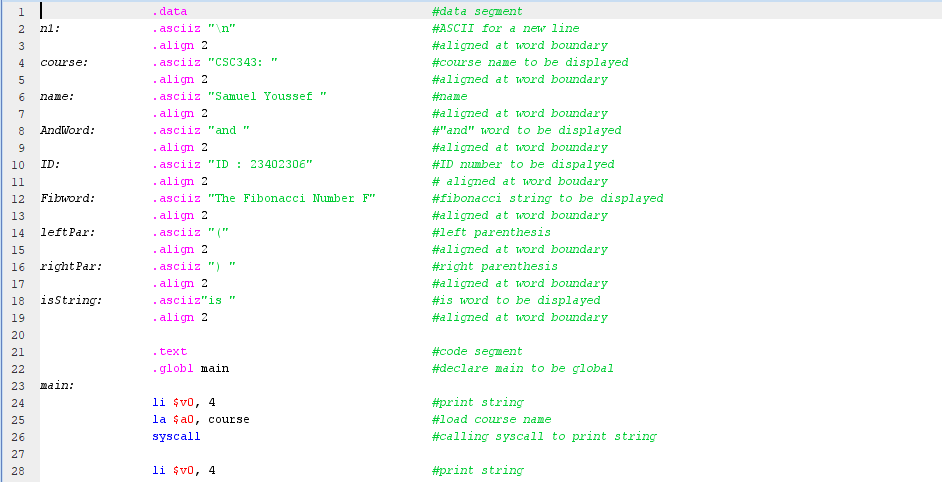


figure 1: the entire source code for task 1.

1. The program outputs are as follows:

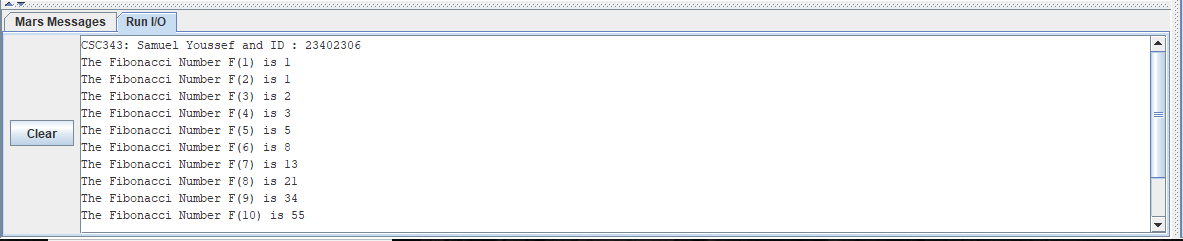


Figure 2: output of task 1.

1. The LA pseudo instruction consists of two instructions:
   1. The LUI (load upper immediate) instruction in figure 3.
   2. The ORI (ors a register value with and immediate value) in figure 4.

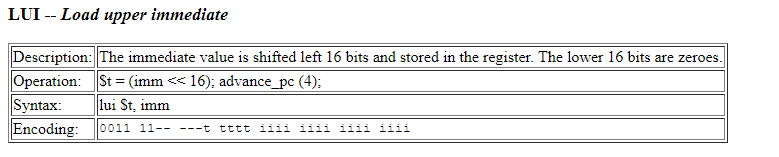


Figure 3: LUI instruction.

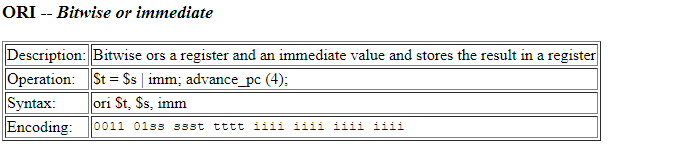


Figure 4: ORI instruction.

🡪Explanation of LA instruction:

* When the LA (pseudo instruction) is used, the assembler converts this instruction into 2 instructions to be executed. The assembler does so because it cannot specify a full 32-bit immediate address value. The first sub instruction is LUI instruction which loads the value of $1 (also named $at (a register reserved for pseudo instructions)) with the base address of the data segment. After that the base address in $at is shifted left by 16 bits; that means that the lower 16 bits of $at are all set to zero. And the upper 16 bits of $at contains the base address. The second sub instruction is the ORI instruction. This instruction ors the value of $at (upper half is the base address and lower half is all zeroes) with another immediate value that contains the offset address of the targeted piece of data in the data segment. After bitwise OR process ends, the destination register in the ORI instruction now contains the full address of the that piece of data.
* **Task 2:**

1. The entire source code of task 2 is:

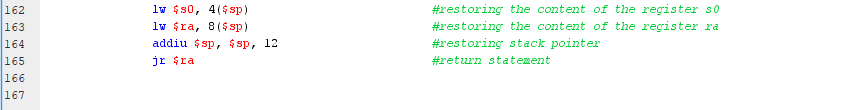
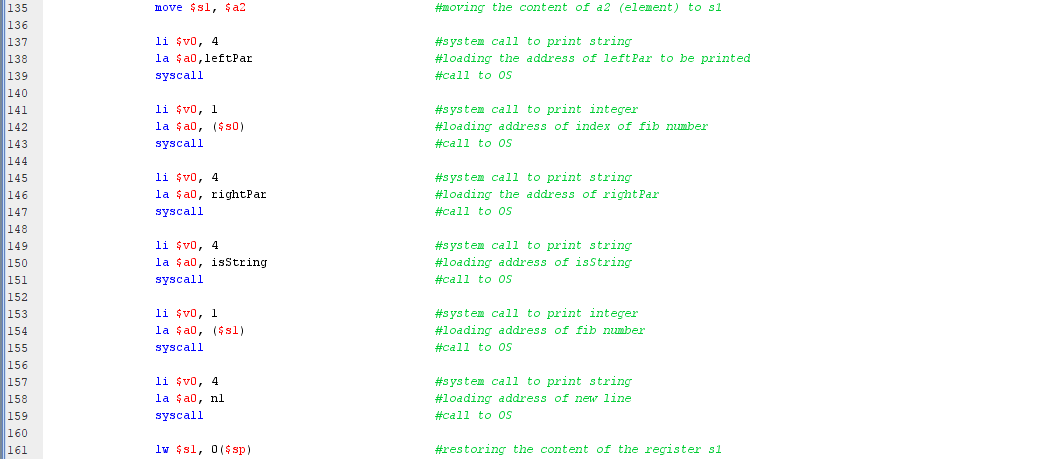
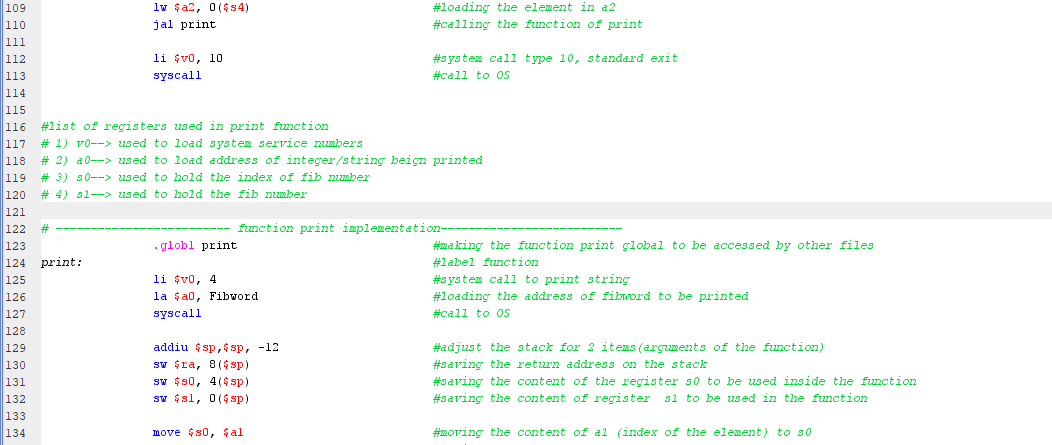
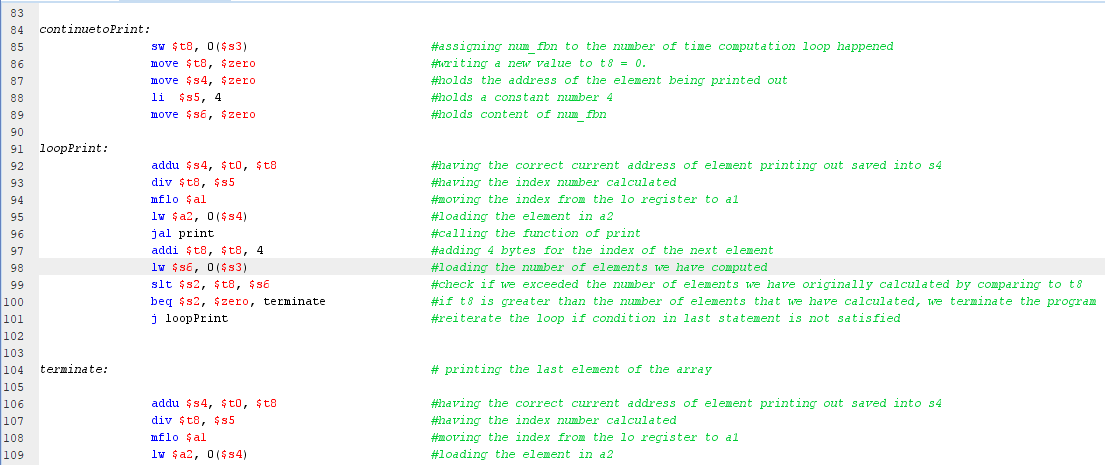
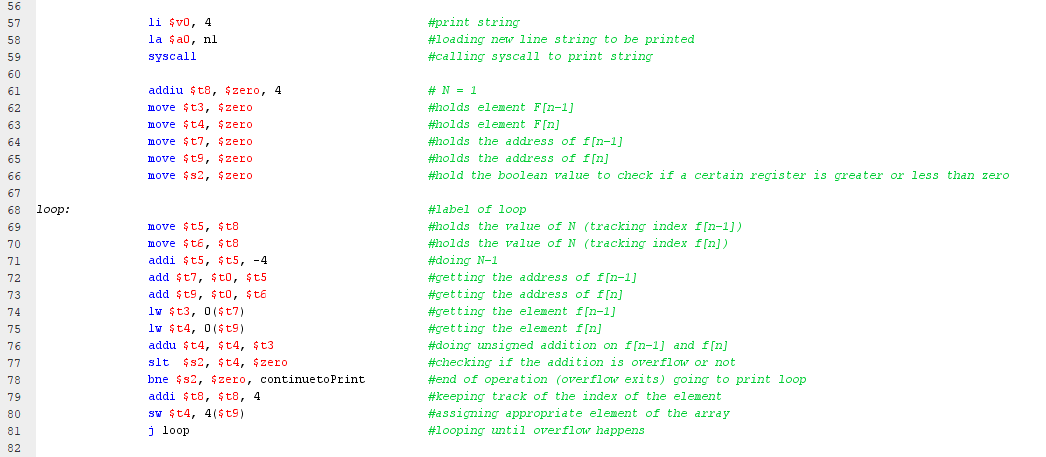
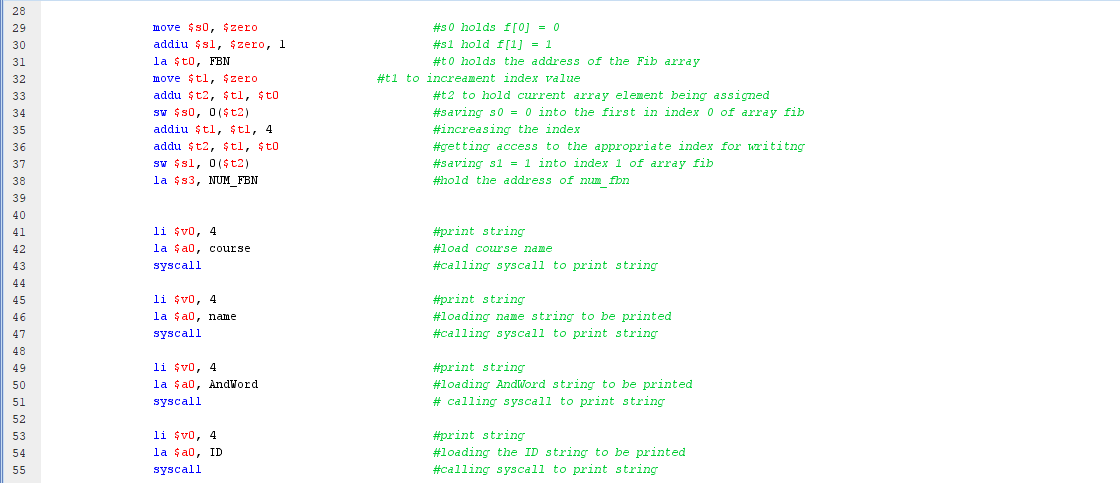
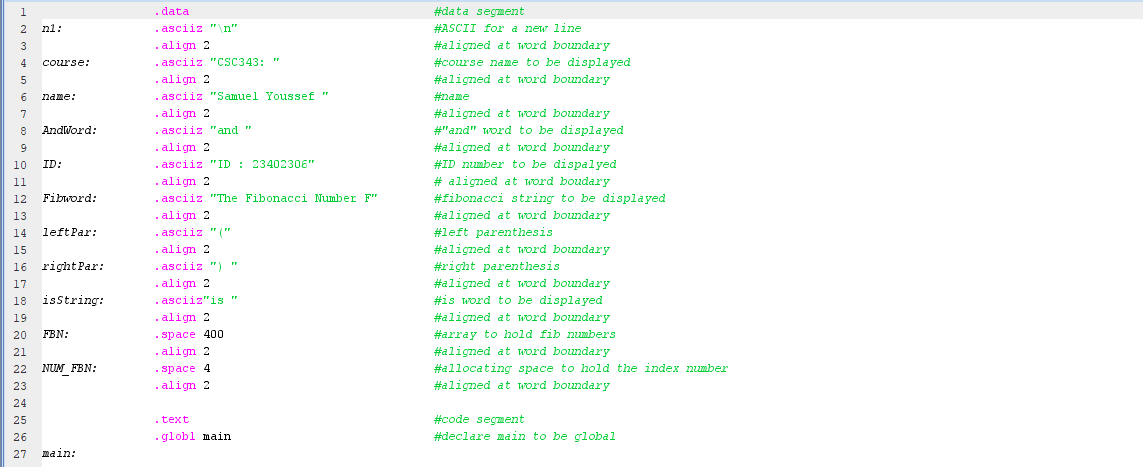


Figure 5: the entire source code of task 2.

1. In this lab, we used the unsigned instructions of addition (addu, and addiu) and we assumed that all numbers are signed. The biggest 32-bit signed binary number is 231 – 1 = 2,147,483,647 (7FFF, FFFF16). We checked if there is an overflow by using ***slt*** instruction as illustrated in figure 6. If the value of addition exceeds the number (231 – 1 = 2,147,483,647), then there is an overflow and the addition of two positive integers generates a negative value. That is the reason why we checked to see if the destination register of the addition process contained a value less than zero or not. Task 2 generated Fibonacci numbers (from index 0 to index 46). Fibonacci number (47th) is 2,971,215,073 which is greater than the biggest signed value a 32-bit register can hold. So obviously the loop terminated after calculating the 47th Fibonacci value ***without*** storing that value in the FBN array.



Figure 6: Statements in Task 2 to check if the addition process result in overflow.

1. The program (of task 2) output is as follows:

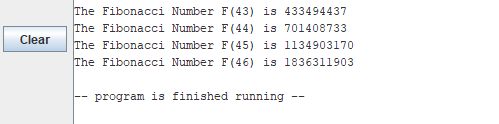
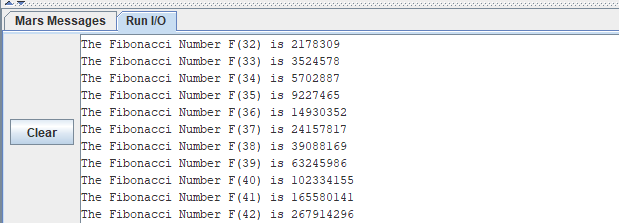
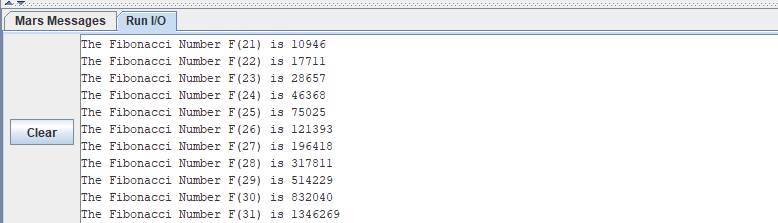
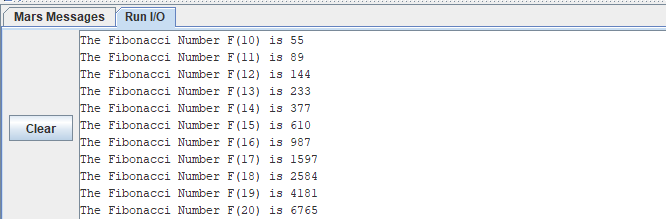
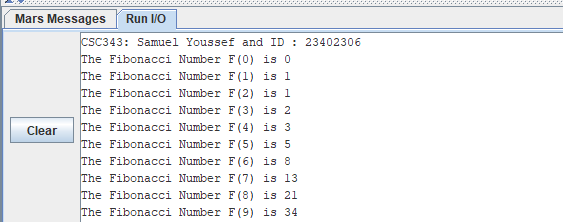


Figure 7: output of task 2.