**LAB REPORT**

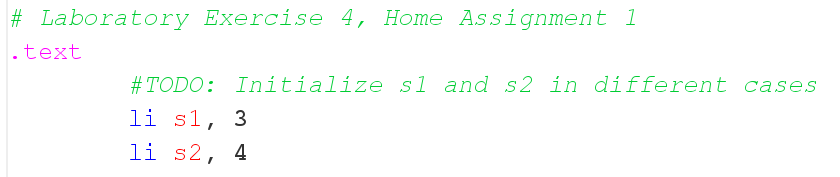
**IT3280E– 152049– Assembly Language and Computer Architecture Lab**

**Lab 04: Arithmetic and Logical Instructions**

# **Assignment 1:**

*Create a new project to implement the Home Assigment 1. Compile and upload to simulator. Initialize two operands (register s1 and s2), run this program step by step, observe memory and registers value.*

* Initialize Operands**:**



+ *s1* = 3 and *s2* = 4

* Step by step:

+ After initializing:





**+** Overflow detection:

* *t0* is set to 0 (no overflow by default)
* the sum of *s1* and *s2* is stored in *s3*
* ***xor***: checks if two operands (*s1*, *s2*) have the same sign
* the code then detects overflow or safely exits without overflow
* since *s3* (= s1 + s2 = 3 + 4 )= 7 > *s1* = 3 => **there is no overflow**

A white rectangular object with a black border

Description automatically generated

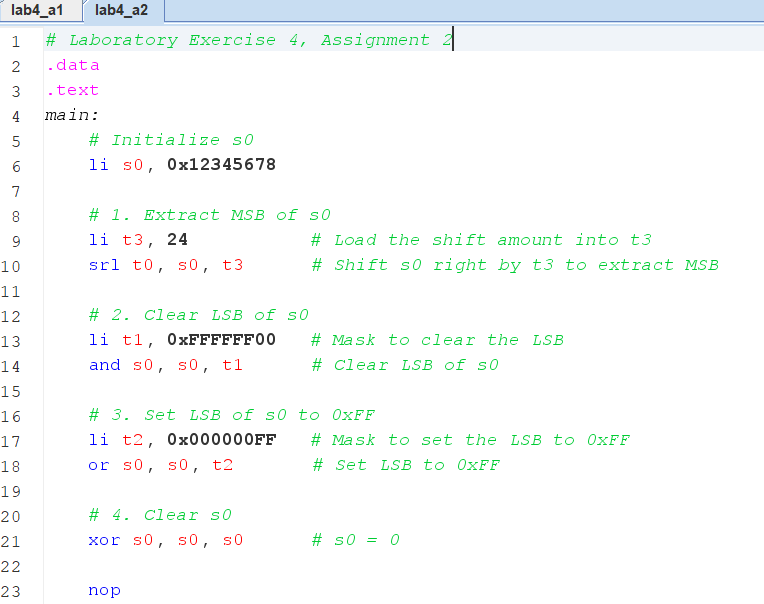
*t0 = 0 (no overflow); t1 = 7 (have same sign), t2 = 0 (both s1, s2 are postive)*

* Key Observations:

+ The memory doesn’t change much because this code mostly operates on registers rather than memory locations

**Assignment 2:**

*Write a program to do the following tasks: ▪ Extract MSB of s0 ▪ Clear LSB of s0 ▪ Set LSB of s0 (bits 7 to 0 are set to 1) ▪ Clear s0 (s0=0, must use logical instructions)*



* Explain Code:

+ **Extract MSB**: Shift *s0* right by 24 bits, storing the MSB in *t0*.

+ **Clear LSB**: Use a mask (*0xFFFFFF00*) and perform bitwise AND to clear the LSB in *s0*.

+ **Set LSB to *0xFF***: Use a mask (*0x000000FF*) and perform bitwise OR to set the LSB.

+ **Clear *s0***: xor *s0* with itself to set its value to 0

# **Assignment 3:**

*Pseudo instructions in RISC-V are not-directly-run-on-RISC-V-processor instructions which need to be converted to real-instructions of RISC-V. Re-write the following pseudo instructions using real-instructions understood by RISC-V processors:*

1. *abs s0, s1*

*s0 = abs(s1)*

A screenshot of a computer

Description automatically generated

1. *move s0, s1*

*s0 = s1*



1. *not s0*

*s0 = bit\_invert(s0)*



1. *ble s1, s2, label*

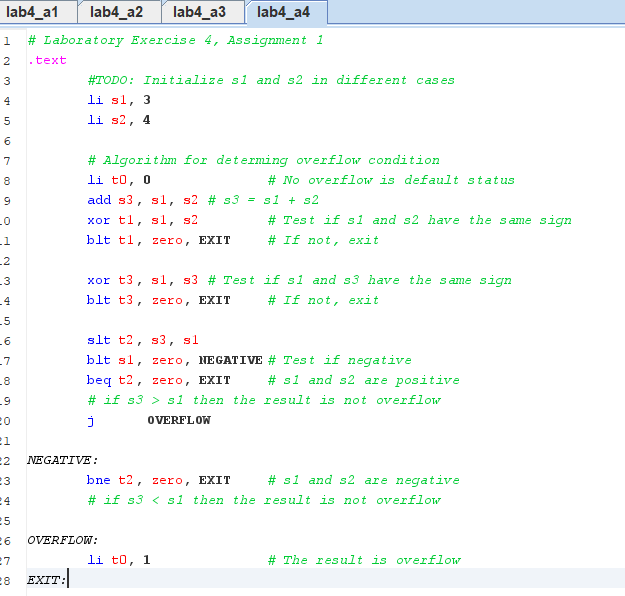
*if(s1 <= s2)*

*j label*



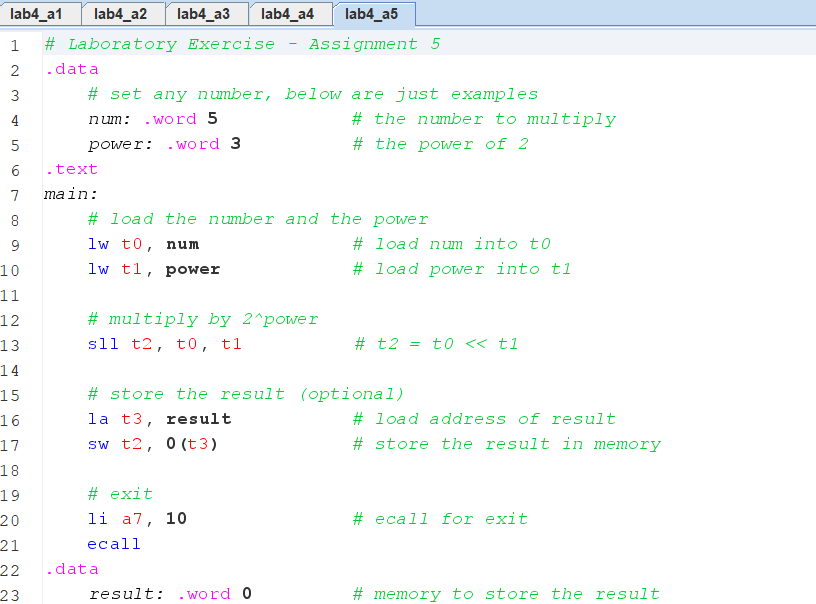
# **Assignment 4:**

*To dectect overflow in additional operation, we also use other rule than the one in Assignment 1. This rule is: when add two operands that have the same sign, overflow will occur if the sum doesn’t have the same sign with either operands. You need to use this rule to write another overflow detection program.*



# **Assignment 5:**

*Write a program that implement multiply by a small power of 2. (2, 4, 8, 16, etc for example).*



* The example above illustrates the multiplication of
* Explaination:

+ load the number and power into the registers *t0* and *t1*

+ left shifting *t0* by *t1* using ***sll*** to perform the multiplication

+ store result in memory

+ exit the program

* **Conclusion**: Using shift instructions offers several benefits over multiply instructions: they are often faster, simpler, and more efficient for multiplying by powers of two. In particular:

**+** It is much easier to understand and implement (especially for powers of two).

**+** Shifts can be executed in a single cycle, while multiplication may takes multiple cycles.

+ Requires fewer resources

+ Can be done without using complex instructions

+ Provide more control over the operation -> beneficial in certain algorithms