**LAB REPORT NO 04**

****

**Computer organization and Architecture (CSE 304L)**

**Fall 2024**

**Submitted By: Naveed Ahmad**

**Registration No: 22PWCSE2165**

**Section: B**

Date: 02/12/2024

**Submitted To:**

**Dr. Ammad Khalil**

**Department of Computer Systems Engineering**

**University of Engineering and Technology**

**TASK 1:**

1. Create a directory to hold the files for this lab. Launch spim and make sure its register pane displays values in hex.
2. Launch your favorite editor and type the following fragment:

.text

main: #---------------------

addi $t0, $0, 60

addi $t1, $0, 7

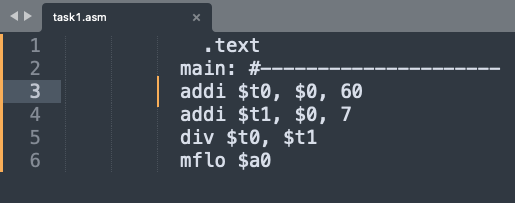
div $t0, $t1

mflo $a0

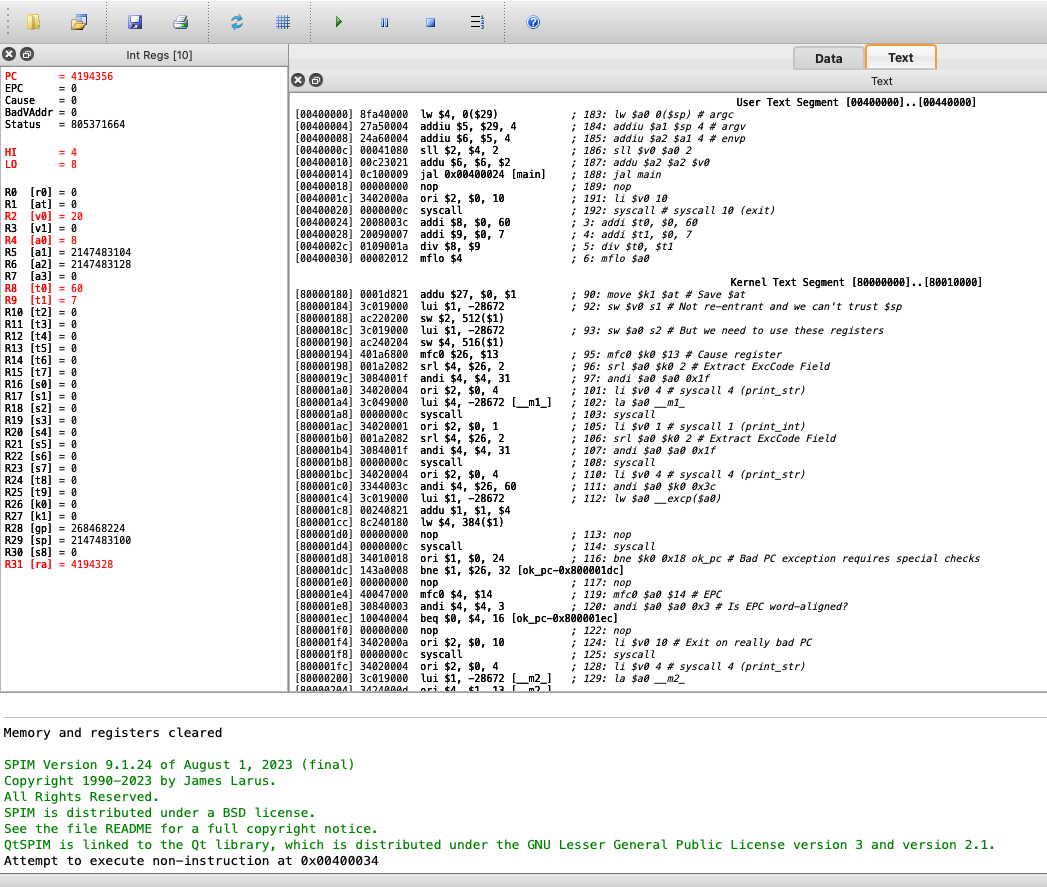
The div instruction is equivalent to: lo = $t0 / $t1;  
hi = $t0 % $t1;

In other words, it sets lo register to the *quotient* and hi register to the *remainder* of dividing its two integer operands. Since these two registers are not programmable (i.e. cannot be referenced in a program), the two instructions: mflo (move-from-lo) and mfhi (move-from-hi) were added to the instruction set to facilitate copying the contents of lo and hi to programmable registers. Complete the above fragment by having it output lo and hi. Save the program under the name: Lab4\_1.s. Run the program and verify that it works as expected.

**CODE PIC:**

****

**OUTPUT PIC:**

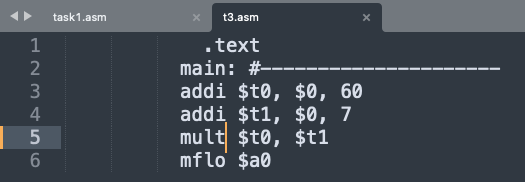
****

3. Modify the program so that it also prints the product of the two integers. The instruction:

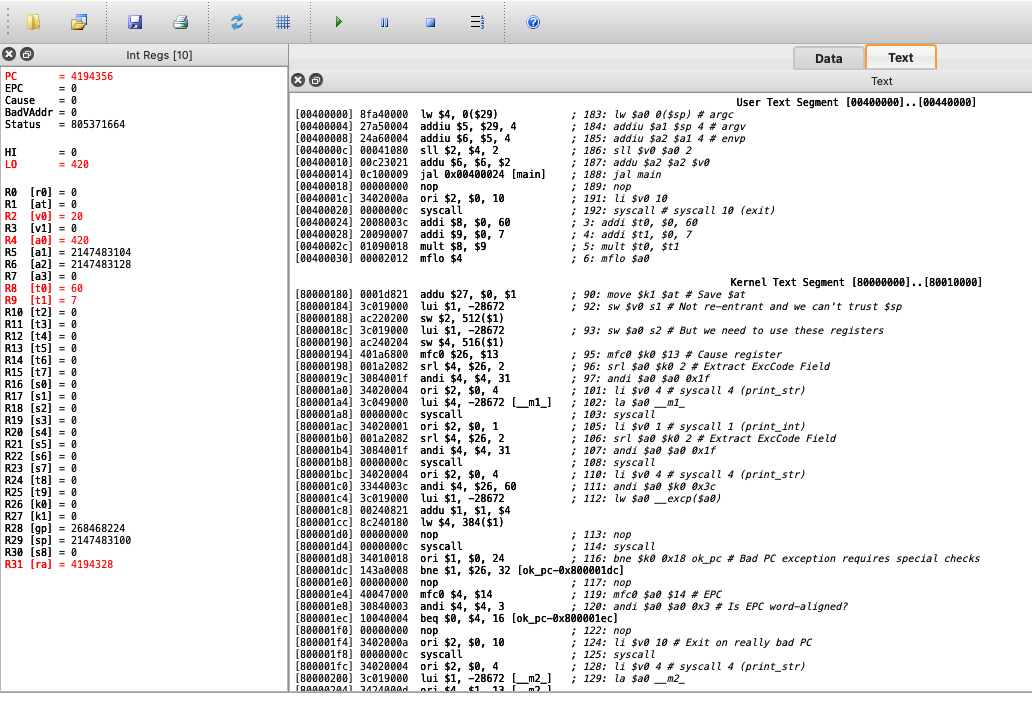
mult $t0, $t1  
multiplies the two operands and store the 64-bit product in hi-lo, i.e. its most

significant 32 bits in hi and its least-significant 32 bits in lo. Again, transfer these two pieces to general-purpose registers and print them.

**CODE PIC:**

****

**OUTPUT PIC:**

****

**TASK 2:**

1. Start fresh and create the program Lab4\_2.s as follows:

.text  
main: #---------------------  
addi $t0, $0, 60  
srl $a0, $t0, 1 # a0 = t0 shifted right once # *print($a0);*# *print(' ');*sll $a0, $t0, 1 # a0 = t0 shifted left once # *print($a0);*#---------------------  
fini: jr $ra

Replace the comments by appropriate output statements. Run the program and examine its two outputs. Explain the output based on your understanding of how integers are represented in binary.

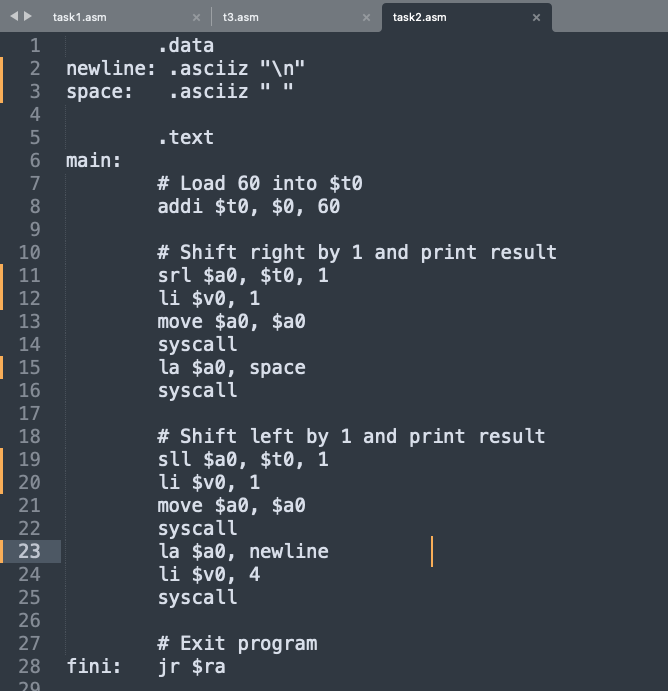
1. Set a breakpoint immediately after the srl instruction. When the program pauses, examine the values of $t0 and $a0 as shown in the register pane. Explain these values based on your understanding of how integers are represented in hex.
2. Change the shift amount from 1 to 2. Re-run the program and interpret its output.
3. Is the left shift sign sensitive? Does it correctly perform multiplication when the

operand is negative?

1. Is the right shift sign sensitive? Does it correctly perform division when the

operand is negative?

**CODE PIC:**

****

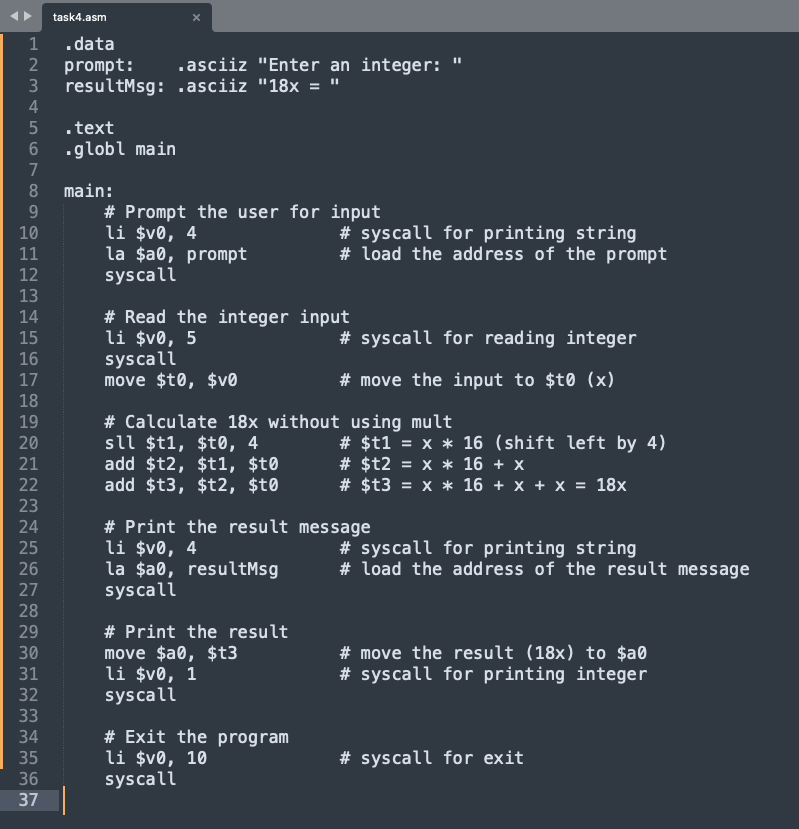
**OUTPUT PIC:**

****

**TASK 3:**

1. Write the program Lab4\_3.s that reads an integer x from the user and outputs 18x *without* using mult. Note that you can rewrite 18x as a sum of two terms each of which multiplies x by a power of 2.
2. Replacing multiplications by shifts is often done in high-level languages by optimizing compilers. Why is shifting a register faster than multiplying it?

**CODE PIC:**

****

**OUTPUT PIC:**

****

**TASK 4:**

11. The program Lab4\_4.s seeks to determine and output the value of bit #10 in $t0. Note that bits are numbered right-to-left starting with zero. It does so by isolating this bit using the following pseudo-code:

*a0 = t0 shifted left 21 times*

*a0 = a0 shifted right 31 times*

Write this program so that it reads an integer into $t0; performs the above algorithm; and then outputs $a0. For example, if the input is 5000, the output should be 0, but if the input were 6000, the output would be 1.

12. Add the following fragment to Lab4\_4.s: andi $a0, $t0, 1024

srl $a0, $a0, 10

syscall

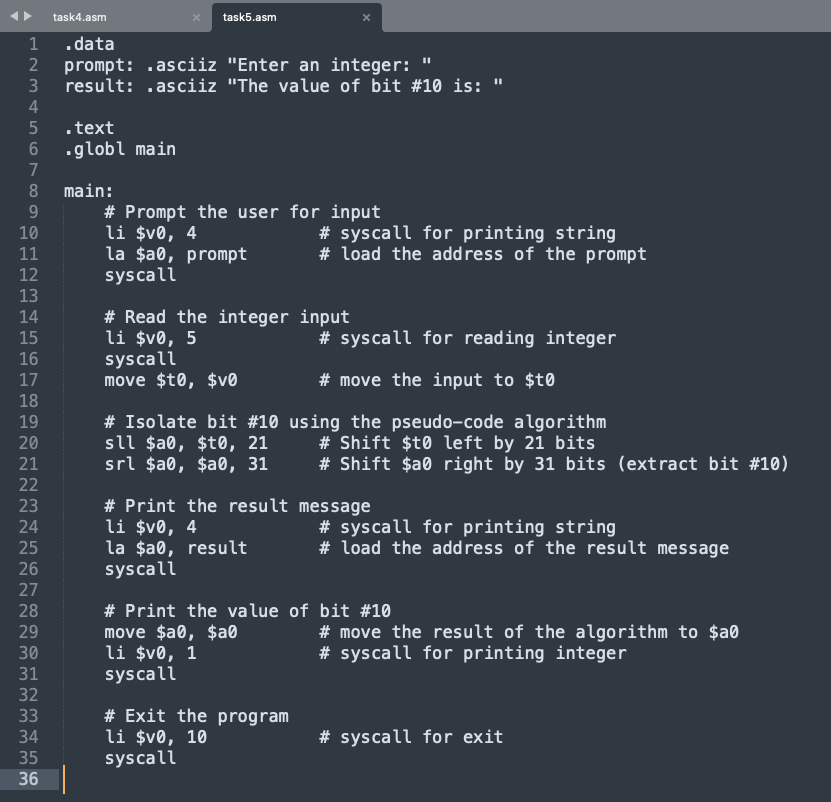
The number, 1024, is known as the **mask** and it has the following representation in binary:

0000 0000 0000 0000 0000 0100 0000 0000

Verify that this mask-based approach will also output the state of bit #10. Could we have written the mask hex?

13. The word "mask" is often used to describe something that blocks the view except for a hole through which one can see. Does the number 1024 behave like a mask in this sense?

**CODE PIC:**



**OUTPUT PIC:**

****

**TASK 5:**

1. We seek to write the program Lab4\_5.s that clears bit #10 in $t0; i.e. sets it to zero. The idea is to read an integer into $t0 and then AND it with the following mask:
2. 1111 1111 1111 1111 1111 1011 1111 1111

Unfortunately this does not work because the immediate in the instruction:

andi $t5, $t0, 0xfffffbff

has to have at most 16 bits. Hence, we construct this mask in stages as follows:

$t5 = 0xffff;

$t5 = $t5 shifted left 16 times;

$t5 = $t5 or 0xfbff;

Implement the program and run it. Use ori instead of addi to store a bit pattern in a register Verify that it works as expected. For example, if the input were 5000 then the output should be 5000 whereas if the input were 6000 then the output should be 4976.

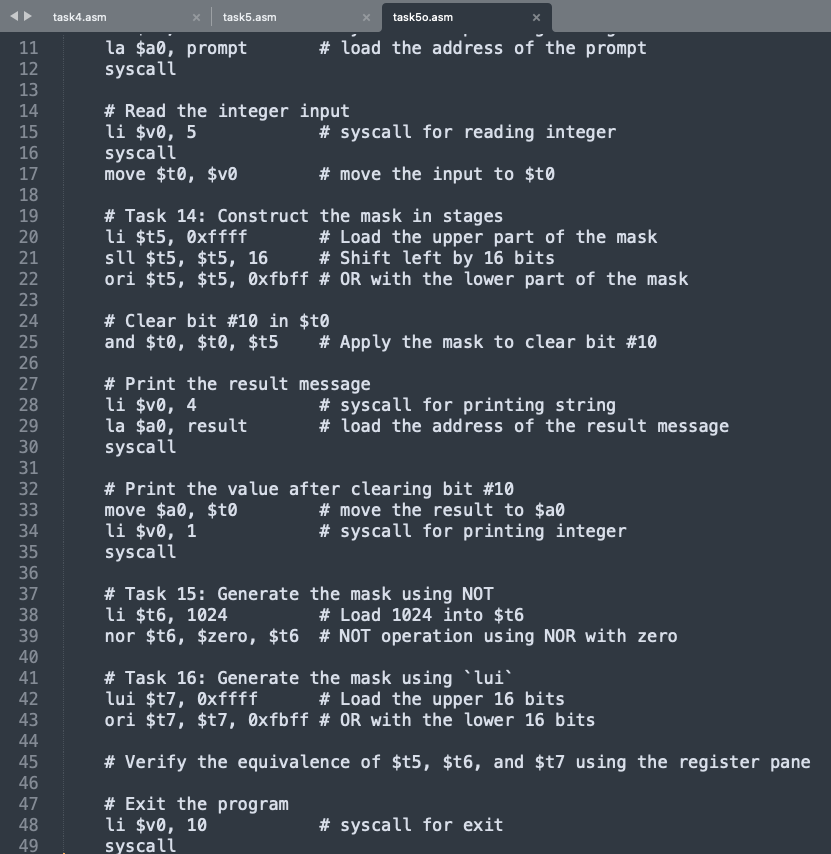
1. The mask of the previous task could also be generated using the following algorithm:

$t6 = 1024;

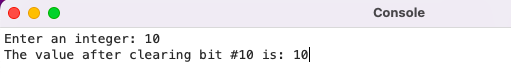
$t6 = not $t6;  
There is no not instruction in MIPS but there is a nor (i.e. not or). Add code to Lab4\_5.s so that it generates the mask using the new method. Are the two methods equivalent? You can simply inspect the register pane of spim and compare the contents of $t5 and $t6.

1. The mask of the previous task could also be generated using the lui instruction. Split the immediate 0xfffffbff into two pieces; send them separately in two different instructions; and then reconstitute the 32-bit immediate.

**CODE PIC:**

****

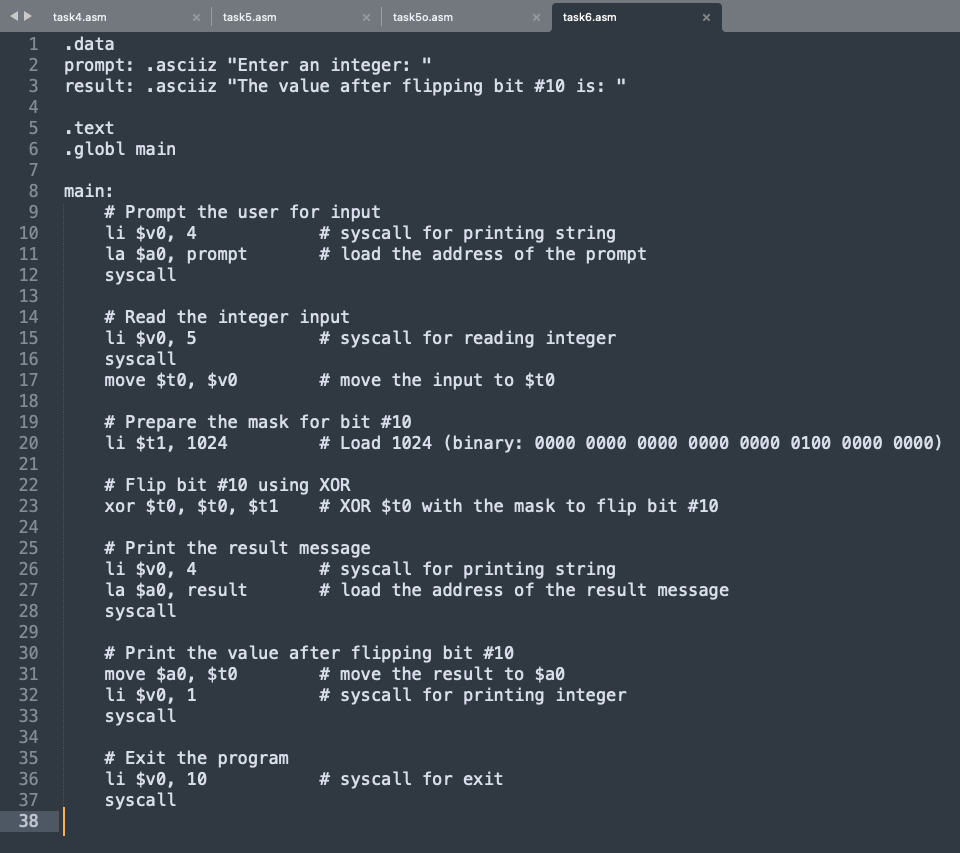
**OUTPUT PIC:**

****

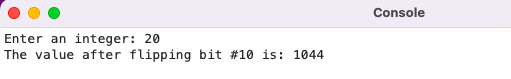
**TASK 6:**

1. It is often said that exclusive-or is useful for detecting differences in bit values, i.e. detecting the combinations 01 and 10. Justify this claim by examining the truth table of xor.
2. Write the program Lab4\_6.s that flips bit #10 in $t0. Prepare an appropriate mask and note that when you xor a bit with 1, you get the negation of that bit, but when you xor with 0, you get the original bit unchanged.

**CODE PIC:**

****

**OUTPUT PIC:**

****