



Department of Computer Science

Lab Manual

CSC 221-L Computer Organization and Assembly Language

Instructor's Name: Muhammad Hanif

Student's Name: Ali-ur-Rehman

Roll No.: 67267 Batch: 2027

Semester: 3rd Year: 2nd

Department: Computer Science

Slot Time & Day: Thrusday (11:30 to 2:25)



FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Department of Computer Science

Lab Manual

CSC 221-L Computer Organization and Assembly Language

Prepared By:

Muhammad Hanif

Reviewed / Approved By:

Faculty of Engineering Sciences & Technology

Iqra University

CONTENTS

Lab. No.	Date	List of Experiment	Total Marks	Signature	Page #
1.		Installing Assembly Language Software (NASM) DOS and getting familiar with its features and operations.			
2.		To perform Addition, Subtraction between two numbers using NASM DOS simulator.			
3.		To perform Multiplication and Division of two numbers using NASM DOS simulator.			
4.		To perform calculations using loops in NASM DOS simulator			
5.		To study the theory of Flag Registers and some of its types.			
6.		Perform addition and multiplication of two numbers by getting values from user using MIPS simulator.			
7.		To store multiple values in an Array using MIPS simulator.			
8.		Mid Term Examination	25		
9.		To perform operations on Floating Points and Double Values using MIPS simulator.			
10.		To perform comparison between numbers and floats using MIPS simulator.			
11.		Using while loop in MIPS simulator.			
12.		Calculating Factorial of a number given by user using MIPS simulator.			
13.		Calculating Average of numbers using MIPS simulator.			
14.		Implement if Statement using Branch			
15.		Open-Ended lab	10		
16.		Open Ended Lab Assesment			
		Final Examination	40		

Psychomotor Rubrics for Software based Lab

Course Name (Course Code): _____

Semester: _____

Criteria	Exceeds Expectations ($\geq 90\%$)	Meets Expectations (70%-89%)	Developing (50%-69%)	Unsatisfactory ($< 50\%$)
Software Skills	Ability to use software with its standard and advanced features without assistance	Ability to use software with its standard and advanced features with minimal assistance	Ability to use software with its standard features with assistance	Unable to use the software
Programming / Simulation	Ability to program/ simulate the lab tasks with simplification	Ability to program/ simulate the lab tasks without errors	Ability to program/ simulate lab tasks with errors	Unable to program/simulate
Results	Ability to achieve all the desired results with alternate ways	Ability to achieve all the desired results	Ability to achieve most of the desired results with errors	Unable to achieve the desired results
Laboratory Manual	All sections of the report are very well written and technically accurate.	All sections of the report are technically accurate.	Few sections of the report contain technical errors.	All sections of the report contain multiple technical errors.

Psychomotor Rubrics Assessment Software based Lab

Course Name (Course Code): _____

Semester: _____

Lab #	Score Allocation				
	Software Skills Marks (3)	Programming/ Simulation Marks (2)	Experimental Results Marks (3)	Laboratory Manual Marks (2)	Total Marks (3)
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
Total Marks		140	Total Obtained marks		

Overall Score: _____ out of 14 Examined by: _____

(Obtained Score / 140) x 14

(Name and Signature of lab instructor)

Affective Domain Rubrics Assessment

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Course Name (Course Code): _____

Semester: _____

CATEGORY	Excellent (100% - 85%)	Good (84% - 75%)	Fair (74% - 60%)	Poor (Less than 60%)
Speaks Clearly	Speaks clearly and distinctly all the time, and confidently.	Speaks clearly and distinctly most of the time, but is confused for a brief period of time, however, recovers.	Speaks clearly and distinctly most of the time, but seems not confident about what has been delivered. Shows lack of confidence.	Often mumbles or cannot be understood and clearly lacks confidence in delivering the content
Points:				
Preparedness	Student is completely prepared and has obviously rehearsed.	Student seems pretty prepared but might have needed a couple more rehearsals.	The student is somewhat prepared, but it is clear that rehearsal was lacking.	Student does not seem at all prepared to present.
Points				
Answer back	Student calmly listens to the questions and responds to the question confidently and correctly	Student calmly listens to the questions, responds confidently but some of the responses are incorrect.	Student shows anxiety while listening to the questions, and gives some correct responses, but some of the responses are incorrect.	Student shows anxiety while listening to the questions, and most of the responses are incorrect.
Points:				
Posture, Eye Contact & Speaking Volume	Stands up straight, looks relaxed and confident. Establishes eye contact with everyone in the room during the presentation. Volume is loud enough to be heard by all members in the audience throughout the presentation.	Stands up straight and establishes eye contact with everyone in the room during the presentation. Volume is loud enough to be heard by the audience, but is sometimes not audible.	Sometimes stands up straight and establishes eye contact. Volume is loud enough to be heard by the audience, but many sentences spoken are not clear.	Lazy and informal posture. Does not look at people during the presentation. Volume is also too soft to be heard by the audience.
Points:				

Overall Score: _____ out of 14 Examined by: _____

(Name and Signature of lab instructor)

Open Ended Lab Assessment Rubrics

Course Name (Course Code): _____

Semester: _____

Criteria and Scales			
Excellent (10-8)	Good (7-5)	Average (4-2)	Poor (1-0)
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student			
Understands the problem clearly and clearly identifies the underlying issues.	Adequately understands the problem and identifies the underlying issues.	Inadequately defines the problem and identifies the underlying issues.	Fails to define the problem adequately and does not identify the underlying issues.
Criterion 2: Research: The amount of research that is used in solving the problem			
Contains all the information needed for solving the problem	Good research, leading to a successful solution	Mediocre research which may or may not lead to an adequate solution	No apparent research
Criterion 3: Class Diagram: The completeness of the class diagram			
Class diagram with complete notations	Class diagram with incomplete notations	Class diagram with improper naming convention and notations	No Class diagram
Criterion 4: Code: How complete and accurate the code is along with the assumptions			
Complete Code according to the class diagram of the given case with clear assumptions	Incomplete Code according to the class diagram of the given case with clear assumptions	Incomplete Code according to the class diagram of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 5: Report: How thorough and well organized is the solution			
All the necessary information clearly organized for easy use in solving the problem	Good information organized well that could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

Open Ended Lab Assessment Rubrics

Course Name (Course Code): _____

Semester: _____

Criteria and Scales				
Excellent (10-8)	Good (7-5)	Average (4-2)	Poor (1-0)	Total Marks 10
<u>Criterion 1:</u> Understanding the Problem: How well the problem statement is understood by the student				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
<u>Criterion 2:</u> Research: The amount of research that is used in solving the problem				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
<u>Criterion 3:</u> Class Diagram: The completeness of the class diagram				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
<u>Criterion 4:</u> Code: How complete and accurate the code is along with the assumptions				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
<u>Criterion 5:</u> Report: How thorough and well organized is the solution				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Total				(___/5)

Total marks obtained: _____

Name and Signature of lab instructor: _____

Rubrics for Lab Project / CCA

Course Name (Course Code): _____

Semester: _____

Criteria	Exceeds Expectations ($\geq 90\%$)	Meets Expectations (70%-89%)	Developing (50%-69%)	Unsatisfactory ($< 50\%$)
Project Presentation + Project Demonstration	Ability to demonstrate the project with achievement of required objectives having clear understanding of project limitations and future enhancements. Hardware and/or Software modules are fully functional, if applicable.	Ability to demonstrate the project with achievement of required objectives but understanding of project limitations and future enhancements is insufficient. Hardware and/or Software modules are functional, if applicable.	Ability to demonstrate the project with achievement of at least 50% required objectives and insufficient understanding of project limitations and future enhancements. Hardware and/or Software modules are partially functional, if applicable.	Ability to demonstrate the project with achievement of less than 50% required objectives and lacks in understanding of project limitations and future enhancements. Hardware and/or Software modules are not functional, if applicable.
Project Report	All sections of the Project report are very well-written and technically accurate.	All sections of the Project report are technically accurate.	Few sections of the Project report contain technical errors.	Project report has several grammatical/spelling errors and sentence construction is poor.
Viva	Able to answer the questions easily and correctly across the project.	Able to answer the questions related to the project	Able to answer the questions but with mistakes	Unable to answer the questions

Total marks: _____

Name and Signature of lab instructor: _____

Project / CCA Rubric based Assessment



FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Course Name (Course Code): _____

Semester: _____

Project #	Score Allocation			
	Project Presentation + Project Demonstration Marks (5)	Project Report Marks (3)	Viva Marks (3)	Total Marks (10)
1				
2				
Total Obtained Score				

Total marks obtained: _____

Name and Signature of lab instructor: _____



FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

Final Lab Assessment

Assessment Tool	CLO-1 (20)	CLO-2 (20)	CLO-3 (10)
Lab Manual			
Subject Project / Viva			
Lab Exam / Viva			
Score Obtained			
Total Score: _____ out of 50			

Examined by: _____

(Name and Signature of concerned lab instructor)

Introduction

Computer Organization and Architecture is the study of internal working, structuring and implementation of a computer system. Architecture in computer system, same as anywhere else, refers to the externally visual attributes of the system. Externally visual attributes, here in computer science, mean the way a system is visible to the logic of programs (not the human eyes!). Organization of computer system is the way of practical implementation which results in realization of architectural specifications of a computer system.[In more general language, Architecture of computer system can be considered as a catalog of tools available for any operator using the system, while Organization will be the way the system is structured so that all those cataloged tools can be used, and that in an efficient fashion.

The Computer Organization and Architecture has following four parts:

1. Memory Organisation
2. Input/Output Unit.
3. Control Unit.
4. Buses.

Assembly Language

This is a brief introduction to assembly language. Assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations implemented directly on the physical . Assembly language lacks high-level conveniences such as variables and functions, and it is not portable between various families of processors. Nevertheless, assembly language is the most powerful computer programming language available, and it gives programmers the insight required to write effective code in high-level languages. Learning assembly language is well worth the time and effort of every serious programmer.

Three different type of Assemblers are used for assembly language.

1. NASM (Netwide Assembler).
2. MASM (Macro Assembler).
3. TASM (Turbo Assembler).

Registers:

Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon. However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus and into the memory storage unit and getting the

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

data through the same channel.

To speed up the processor operations, the processor includes some internal memory storage locations, called **registers**.

The registers store data elements for processing without having to access the memory.

A limited number of registers are built into the processor chip.

Processor Registers

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories –

- ☐ General registers,
- ☐ Control registers, and
- ☐ Segment registers.

The general registers are further divided into the following groups –

- ☐ Data registers,
- ☐ Pointer registers, and
- ☐ Index registers.

Data Registers

Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways –

- ☐ As complete 32-bit data registers: EAX, EBX, ECX, EDX.
- ☐ Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX. ☐

Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.

Some of these data registers have specific use in arithmetical operations.

AX is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.

FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY

BX is known as the base register, as it could be used in indexed addressing.

CX is known as the count register, as the ECX, CX registers store the loop count in iterative operations.

DX is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

Pointer Registers

The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers –

□

Instruction Pointer (IP) – The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment. □

Stack Pointer (SP) – The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack. □

Base Pointer (BP) – The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.

Index Registers

The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction.

There are two sets of index pointers –

□ **Source Index (SI)** – It is used as source index for string operations.

□ **Destination Index (DI)** – It is used as destination index for string operations.

General Instructions:

- **MOV:** Destination source
- **POP:** Destination
- **PUSH:** Source
- **XCHG:** Destination

Equipments:

Two main equipments are used for assembly language:

1. **NASM (Netwide Assembler).**

DOSBox 0.74, Cpu speed: 3000 cycles, Frame...

Register	Value	Register	Value	Register	Value	Register	Value	Stack	Flags
AX	0000	SI	0000	CS	19F5	IP	0100	+0	0000
BX	0000	DI	0000	DS	19F5			+2	20CD
CX	001C	BP	0000	ES	19F5	HS	19F5	+4	9FFF
DX	0000	SP	FFFE	SS	19F5	FS	19F5	+6	EA00

CMD >	1	0	1	2	3	4	5	6	7
0100 B90A00	MOV	CX,000A	DS:0000	CD	20	FF	9F	00	EA
0103 B80000	MOV	AX,0000	DS:0008	AD	DE	1B	05	C5	06
0106 66	DB	66	DS:0010	18	01	10	01	18	01
0107 B80100	MOV	AX,0001	DS:0018	01	01	01	00	02	FF
010A 0000	ADD	[BX+SI],AL	DS:0020	FF	FF	FF	FF	FF	FF
010C 66	DB	66	DS:0028	FF	FF	FF	FF	EB	19
010D BB0000	MOV	BX,0000	DS:0030	A2	01	14	00	18	00
0110 0000	ADD	[BX+SI],AL	DS:0038	FF	FF	FF	FF	00	00
			DS:0040	05	00	00	00	00	00
			DS:0048	00	00	00	00	00	00

2	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
DS:0000	CD	20	FF	9F	00	EA	F0	FE	AD	DE	1B	05	C5	06	00	00
DS:0010	18	01	10	01	18	01	92	01	01	01	00	02	FF	FF	FF	FF
DS:0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	EB	19	C0	11
DS:0030	A2	01	14	00	18	00	F5	19	FF	FF	FF	FF	00	00	00	00
DS:0040	05	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

1 Step 2ProcStep 3Retrieve 4Help ON 5BRK Menu 6 7 up 8 dn 9 le 10 ri



FACULTY OF ENGINEERING SCIENCES AND TECHNOLOGY
MIPS (Microprocessor without interlocked Pipeline)

mips1.asm - MARS 4.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Full-screen Step

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$a4	8	0x00000000
\$t0	9	0x00000000
\$t1	10	0x00000000
\$t2	11	0x00000000
\$t3	12	0x00000000
\$t4	13	0x00000000
\$t5	14	0x00000000
\$t6	15	0x00000000
\$t7	16	0x00000000
\$t8	17	0x00000000
\$t9	18	0x00000000
\$s0	19	0x00000000
\$s1	20	0x00000000
\$s2	21	0x00000000
\$s3	22	0x00000000
\$s4	23	0x00000000
\$s5	24	0x00000000
\$s6	25	0x00000000
\$s7	26	0x00000000
\$s8	27	0x00000000
\$s9	28	0x00000000
\$gp	29	0xffffffff
\$fp	30	0x00000000
\$ra	31	0x00000000
\$lo		0x00400000
\$hi		0x00000000
\$co		0x00000000

Line: 1 Column: 1 Show Line Numbers

Mars Messages Run IO

Clear

Search the web and Windows

8:04 Ahsan 03-Dec-19

Lab Session 1

Objective:

Installing Assembly Language Software (NASM) and getting familiar with its features and operations.

Required Equipment / tools:

DOSBox

Introduction:

The Netwide Assembler (NASM) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. NASM is considered to be one of the most popular assemblers for Linux. NASM was originally written by Simon Tatham with assistance from Julian Hall.

Procedure:

Steps to Install NASM:

- Assembly Language-Setup (DOSBox, AssmSoft)
- Extract files + press ok.
- Open the folder.
- First DOSBox (Install it).
- Assm Software folder (cut it and install in different drive).
- Then Extract file in that drive and press ok.
- Now open that Assm Soft folder.
- AFD (Advance Full Debugger).
- Run (AFD) in DOSBox.
- Change the name of folder (AssSoft) into Assembly.
- Now open DOSBox and write:
 - A:\> mount b Drive f:\assembly
(Drive b is mounted as local directory)
f:\assembly
 - A:\>b :
 - B: \>nasm abc.asm -o abc.com
 - B: \>AFD ABC.COM

Task #1

Compiled an assembly file using the command `nasm abc.asm -o abc.com`. Explain the purpose of this command and the significance of each part (i.e., `nasm`, `abc.asm`, and `-o abc.com`).

1. Explanation of the Command:

The command `nasm abc.asm -o abc.com` is used to assemble an assembly language file and generate an executable file.

- `nasm`: This is the assembler, which translates the assembly code into machine code.
- `abc.asm`: This is the assembly source file that contains the code to be compiled.
- `-o abc.com`: This specifies the output file. The `-o` option tells NASM to generate a `.com` file, which is a simple executable format for DOS.

Task #2

Write an assembly language program named `abc.asm` that prints the message "Hello from NASM". Compile the program using NASM and generate an executable `.com` file.

```
section .data
```

```
msg db 'Hello from NASM',0
```

```
section .text
```

```
global _start
```

```
_start:
```

```
mov ah, 09h
```

```
lea dx, [msg]
```

```
int 21h
```

```
mov ah, 4Ch
```

```
int 21h
```

Task #3

Write NASM Assembly program that prints your **name**, **roll number**, and **department** to the console.

```
section .data
```

```
name db 'Name: John Doe', 0
```

```
roll db 'Roll No: 12345', 0
```

```
dept db 'Department: CS', 0
```

```
section .text
```

```
global _start
```

```
_start:
```

```
mov ah, 09h
```

```
lea dx, [name]
```

```
int 21h
```

```
lea dx, [roll]
```

```
int 21h
```

```
lea dx, [dept]
```

```
int 21h
```

```
mov ah, 4Ch
```

```
int 21h
```

Discussion and analysis of results:

In this experiment, we successfully installed the NASM assembler within a DOSBox environment and executed a series of assembly programs to demonstrate basic operations. The following key points emerged from our exploration:

1. **Installation Success:** The installation process for both DOSBox and NASM was straightforward. We verified the installation by checking the NASM version, confirming that our setup was correct.
2. **Program Execution:** The three simple assembly programs executed as intended. The "Hi" program successfully printed a string to the console, demonstrating basic output operations. The addition program confirmed the ability to perform arithmetic operations in assembly language, while the counting program illustrated loop control and decremented the counter until it reached zero.
3. **Understanding of Assembly Language:** These exercises reinforced fundamental concepts of assembly programming, including the use of registers, system calls for output, and control structures. The clarity of assembly language syntax and the low-level control it provides over hardware resources were notable advantages.
4. **Error Handling and Debugging:** Throughout the process, any encountered issues (e.g., syntax errors) were promptly identified and resolved, emphasizing the importance of debugging in assembly programming. Tools like AFD proved beneficial for stepping through code execution.

Conclusion:

The experiment provided a valuable opportunity to gain hands-on experience with assembly language programming using NASM in a DOSBox environment. By successfully installing the necessary software and executing basic programs, we developed a foundational understanding of assembly language principles and operations. The results demonstrated not only the functionality of the assembler but also highlighted the critical role of debugging and problem-solving skills in programming. As we move forward, this experience serves as a stepping stone for more complex assembly language projects and applications.

Objective:

Required Equipment / tools:

- ## Introduction:

Procedure

```
[ORG 0X100]
MOV AX,5;
MOV BX,4;
ADD AX,BX;
MOV AX,0X4C00;
INT 0X21
```



➤ **Subtraction Program:**

```
[ORG 0X100]
MOV AX,5;
MOV BX,4;
ADD AX,BX;
MOV BX,2;
SUB AX,BX;
MOV AX,0X4C00;
INT 0X21
```

Task #1

What are the values stored in result_add and result_sub after executing the program with the initial values of num1 (10) and num2 (5)? Explain how these values are computed in the code.

```
section .data
    num1 db 10
    num2 db 5
    result_add db 0
    result_sub db 0

section .text
    mov al, [num1]
    add al, [num2]
    mov [result_add], al

    mov al, [num1]
    sub al, [num2]
    mov [result_sub], al
```

Task #2

Take 3 numbers and first add them then take another number and subtract and store the value in BX register

```
section .data
    num1 db 3
    num2 db 7
    num3 db 2
    num4 db 4

section .text
    mov al, [num1]
    add al, [num2]
    add al, [num3]
    sub al, [num4]
    mov bx, ax
```

Task #3

Modify the program to include error handling for negative numbers. What changes would you need to implement, and how would they affect the output?

```
section .data
    num1 db 3
    num2 db 7
    num3 db 2
    num4 db 4
    msg_error db 'Error: Negative Result', 0

section .text
    mov al, [num1]
    add al, [num2]
    add al, [num3]
    sub al, [num4]
    js error

    mov bx, ax
    jmp end

error:
    lea dx, [msg_error]
    mov ah, 09h
    int 21h

end:
```

Discussion and analysis of results:

In this lab, we successfully implemented a MIPS assembly program that performs addition and subtraction on two integers. The results showed that the program accurately calculated and displayed the addition of 10 and 5 as 15, and the subtraction as 5. The effective use of registers and system calls highlighted the efficiency of assembly language for low-level operations. Additionally, modifying input values demonstrated the program's flexibility, while understanding the MIPS instructions reinforced our grasp of how operations are executed at the hardware level.

Conclusion:

The lab provided valuable insights into assembly language programming, particularly in handling arithmetic operations. By developing the addition and subtraction program, we gained hands-on experience with MIPS syntax and system calls. The ability to modify inputs and format outputs further enhanced our understanding of how to manipulate data in assembly language. Overall, this exercise serves as a solid foundation for exploring more complex programming concepts in future labs.

Lab Session 3

Objective:

To perform Multiplication and Division of two numbers using NASM simulator.

Required Equipment / tools:

1. DOSBOX

Introduction:

In this lab session, the primary objective is to implement multiplication and division operations using the NASM simulator. The multiplication program demonstrates the process of multiplying two values, with an initial addition step to set the correct operands. The division program performs a simple division of two numbers, ensuring proper register setup for division in assembly language. Students will gain hands-on experience with fundamental arithmetic operations, familiarizing themselves with the workings of the NASM environment, register handling, and interrupt usage for program termination. This session builds core skills for developing more complex assembly programs.

Procedure:

➤ Multiplication Program:

```
[ORG 0X100]
MOV AX,5;
MOV BX,4;
ADD AX,BX;
MOV BX,2;
MUL BX;
MOVAX,0X4C00;
INT 0X21
```

➤ Division Program:

```
[ORG 0X100]
MOV DX,0;
MOV AX,25;
MOV BX, 5;
DIV BX;
MOV AX,0X4C00;
INT 0X21
```

➤ Output:

The screenshot shows the DOSBox 0.74 interface. The top status bar indicates 'Cpu speed: 3000 cycles, Frame...'. The main window is divided into several sections:

- Registers:** AX: 4C00, SI: 0000, CS: F000, IP: 14A1, Stack: +0 42BD, Flags: 7200. BX: 0004, DI: 0000, DS: 19F5. CX: 000F, BP: 0000, ES: 19F5, HS: 19F5. DX: 0000, SP: FFF2, SS: 19F5, FS: 19F5.
- Command Line:** CMD >
- Assembly Code:**

```

4A0 FB STI
4A1 FE DB
4A2 3B25 CMP DI, AH
4A4 00CF ADD BH, CL
4A6 CB RET Far
4A7 51 PUSH CX
4A8 B94001 MOV CX, 0140
4AB E2FE LOOP 14AB
4AD 59 POP CX

```
- Memory Dump:**

Address	0	1	2	3	4	5	6	7
DS:0000	CD	20	FF	9F	00	EA	F0	FE
DS:0008	AD	DE	1B	05	C5	06	00	00
DS:0010	18	01	10	01	10	01	92	01
DS:0018	01	01	01	00	02	FF	FF	FF
DS:0020	FF	FF	FF	FF	FF	FF	FF	FF
DS:0028	FF	FF	FF	FF	EB	19	C0	11
DS:0030	A2	01	14	00	18	00	F5	19
DS:0038	FF	FF	FF	FF	00	00	00	00
DS:0040	05	00	00	00	00	00	00	00
DS:0048	00	00	00	00	00	00	00	00

Task #1

Write an assembly program using NASM to multiply two numbers, 8 and 6. Display the result using the appropriate registers and interrupts. Ensure correct register setup before the multiplication.

```

section .data
    msg db 'Result: $'

section .text
    global _start

_start:
    mov ax, 8
    mov bx, 6
    mul bx
    mov cx, ax
    lea dx, [msg]
    mov ah, 09h
    int 21h
    mov ax, cx
    mov ah, 02h
    int 21h
    mov ah, 4Ch
    int 21h

```

Task #2

Create a NASM assembly program to divide the number 36 by 6. Use the appropriate registers to store the dividend and divisor, perform the division, and display the quotient using an interrupt.

```
section .data
    msg db 'Result: $'

section .text
    global _start

_start:
    mov ax, 36
    mov bx, 6
    div bx
    mov cx, ax
    lea dx, [msg]
    mov ah, 09h
    int 21h
    mov ax, cx
    mov ah, 02h
    int 21h
    mov ah, 4Ch
    int 21h
```

Task #3

Develop a NASM program that first adds two numbers, 7 and 3, and then multiplies the result by 5. After that, divide the final product by 4. Display the final result using the appropriate interrupts.

```
section .data
    msg db 'Result: $'

section .text
    global _start

_start:
    mov ax, 7
    add ax, 3
    mov bx, 5
    mul bx
    mov cx, ax
    mov dx, 4
    div dx
    lea dx, [msg]
    mov ah, 09h
    int 21h
```

```
mov ax, cx  
mov ah, 02h  
int 21h  
mov ah, 4Ch  
int 21h
```

Discussion and analysis of results:

In this section, students will analyze the results of the multiplication and division tasks by examining the values stored in the registers after execution. The focus will be on verifying the correctness of the arithmetic operations and understanding how NASM handles these processes at the assembly level. Discussion should include how different registers are utilized for handling operands, storing results, and the significance of initializing specific registers like DX for division.

Conclusion:

This lab provided a practical introduction to arithmetic operations in MIPS assembly language through a multiplication and division program. By working with system calls and manipulating data in registers, we reinforced our understanding of how assembly language operates at a fundamental level. The successful execution of the program sets a strong foundation for future exploration of more complex assembly language concepts and operations.

Lab Session 4

Objective:

The objective of this lab session is to perform calculations using loops in NASM simulator.

Required Equipment / tools:

1. DOSBOX

Introduction:

In NASM, the Count Register (CX) plays a crucial role in managing loops. It is used to store the number of iterations the loop will run. To create a loop, the desired count is first loaded into the CX register, which then decrements with each iteration until it reaches zero, signaling the loop's end. This method ensures efficient looping without needing additional comparisons or conditions. Utilizing the CX register is essential for controlling repetitive operations, making it a fundamental concept in assembly programming for tasks like array processing and repeated calculations.

Procedure:

➤ Simple Loop program:

```
[ORG 0X100]
MOV CX,10;
MOV AX,0;
L1:
ADD AX,1;
LOOP L1
MOV AX, 0X4C00;
INT 0X21
```

➤ Create Fibonacci series using loops:

```
[ORG 0X100]
MOV CX,6;
MOV AX,0;
MOV BX,1;
MOV DX,AX;
L1:
ADD BX,DX;
MOV AX,BX;
MOV BX,DX;
MOV DX,AX;
SUB AX,1;
```

```
LOOP L1  
MOV AX,0X4C00  
INT 0X21
```

Task #1

Write a NASM program to print the numbers from 1 to 20 using a loop.

```
[ORG 0x100]  
  
MOV CX, 20  
MOV AX, 1  
  
L1:  
    ADD AX, '0'  
    MOV DL, AL  
    MOV AH, 02h  
    INT 21h  
    MOV DL, ''  
    MOV AH, 02h  
    INT 21h  
  
    INC AX  
    LOOP L1  
  
MOV AX, 0x4C00  
INT 0x21
```

Task #2

Create a NASM program to print the even numbers from 2 to 20 using a loop.

```
[ORG 0x100]  
  
MOV CX, 10  
MOV AX, 2  
  
L1:  
    ADD AX, '0'  
    MOV DL, AL  
    MOV AH, 02h
```

```
INT 21h
MOV DL, ''
MOV AH, 02h
INT 21h

ADD AX, 2
LOOP L1

MOV AX, 0x4C00
INT 0x21
```

Task #3

Write a NASM program to print the numbers from 50 to 1 in reverse order using a loop.

```
[ORG 0x100]

MOV CX, 10
MOV AX, 2

L1:
    ADD AX, '0'
    MOV DL, AL
    MOV AH, 02h
    INT 21h
    MOV DL, ''
    MOV AH, 02h
    INT 21h

    ADD AX, 2
    LOOP L1

MOV AX, 0x4C00
INT 0x21
```

Discussion and analysis of results:

In this section, students will analyze how loops were implemented to perform simple counting tasks in NASM. The focus will be on understanding how the CX register controls loop iterations and how decrementing CX affects loop execution. Students should verify the correctness of the

output by comparing it to the expected results for each task, and identify potential areas where looping structures could be optimized. This analysis reinforces the importance of the CX register in managing repetitive operations in assembly language programs.

Conclusion:

Loops in NASM, controlled by the CX register, provide an efficient way to perform repetitive tasks like counting or generating sequences. Through this lab, students gained practical experience in setting up and using loops to solve simple problems, highlighting the significance of proper loop management in assembly programming.

Lab Session 5

Objective:

The objective of this lab is to explore the theory of Flag Registers in assembly language, focusing on their functions and types. Understanding of how Flag Registers influence the execution of conditional operations and overall program flow.

Required Equipment / tools:

(1) DOSBOX

Introduction:

Flag Register:

It is a register that contains the current state of the processor and it is 1 bit register, so either you will get 0 or 1 in the Flag Register's value.

Some types of Flag Register are:

Auxiliary Carry Flag (AF):

It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.

Carry Flag (CF):

It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

Zero Flag (ZF):

It indicates the result of an arithmetic or comparison operation. A non-zero result clears the zero flag to 0, and a zero result sets it to 1, signaling that the last operation yielded a result of zero. This flag is essential for decision-making in conditional jumps and branching within assembly programs.

Procedure:

NASM program that demonstrates the use of various flag registers, including the Carry Flag (CF), Auxiliary Carry Flag (AF), and Zero Flag (ZF). This program performs some arithmetic operations and checks the status of the flags after each operation.

```
[ORG 0x100] ; Origin directive
```

```
; Initialize values  
MOV AL, 10  
MOV BL, 5
```

```
; Addition  
ADD AL, BL  
TEST AL, AL  
JZ ZeroFlagSet  
ADC AL, 0  
JC CarryFlagSet
```

```
MOV AL, 0x0F  
ADD AL, 0x01
```

```
MOV AX, 0x4C00  
INT 0x21
```

```
ZeroFlagSet:  
JMP EndProgram
```

```
CarryFlagSet:  
JMP EndProgram
```

```
EndProgram:  
MOV AX, 0x4C00 ; Terminate program  
INT 0x21
```

Task #1

Write a program that performs subtraction using the AL and BL registers and checks the status of the Zero Flag (ZF) and Carry Flag (CF) after the operation.

```
[ORG 0x100]

MOV AL, 10
MOV BL, 5
SUB AL, BL

; Check Zero Flag (ZF)
MOV AH, 02h
MOV DL, ZF
INT 21h

; Check Carry Flag (CF)
MOV AH, 02h
MOV DL, CF
INT 21h

MOV AX, 0x4C00
INT 0x21
```

Task #2

Create a program that adds two 8-bit numbers stored in registers and examines the Auxiliary Carry Flag (AF) to determine if there was a carry from bit 3 to bit 4 during the addition.

```
[ORG 0x100]

MOV AL, 5
MOV BL, 3
ADD AL, BL

; Check Auxiliary Carry Flag (AF)
MOV AH, 02h
MOV DL, AF
INT 21h

MOV AX, 0x4C00
INT 0x21
```

Task #3

Develop a program that compares two numbers using the CMP instruction and records the status of the Zero Flag (ZF) and Carry Flag (CF) to determine if they are equal or if one is greater than the other.

```
[ORG 0x100]

MOV AL, 10
MOV BL, 5
CMP AL, BL

; Check Zero Flag (ZF)
MOV AH, 02h
MOV DL, ZF
INT 21h

; Check Carry Flag (CF)
MOV AH, 02h
MOV DL, CF
INT 21h

MOV AX, 0x4C00
INT 0x21
```

Discussion and analysis of results:

In this section, students will discuss and analyze the results obtained from the practical tasks involving Flag Registers. They will evaluate how different arithmetic operations affect the status of flags such as the Zero Flag (ZF), Carry Flag (CF), and Auxiliary Carry Flag (AF). By observing the flag states after each operation, students will gain insights into how these flags influence program flow and decision-making in assembly language.

Conclusion:

The lab activities demonstrated the critical role of Flag Registers in assembly programming. By effectively managing and interpreting flag statuses, students learned how to control program execution based on arithmetic results. This foundational understanding of flags enhances their capability to develop more complex assembly language programs that require conditional logic.

Lab Session 6

Objective:

The objective of this lab is to perform addition and multiplication of two numbers entered by the user using the MIPS simulator. Students will implement programs that accept user input, process the values, and output the results of the arithmetic operations.

Required Equipment / tools:

(1) MARS MIPS Simulator

Introduction:

In this lab, students will work with the MIPS simulator to develop assembly programs that interact with the user by obtaining input values. The program will then perform addition and multiplication on these values and display the results. This exercise will help students understand how to handle user input, manipulate data in registers, and perform basic arithmetic operations in the MIPS assembly language.

Procedure:

➤ Addition & Multiplication Program:

INPUT:

```
.data
prompt1: .asciiz "Enter the first number: "
prompt2: .asciiz "Enter the second number: "
result_add: .asciiz "The result of addition is: "
result_mult: .asciiz "The result of multiplication is: "
newline: .asciiz "\n"
.text
.globl main

main:

    li $v0, 4
    la $a0, prompt1
    syscall
    li $v0, 5
    syscall
    move $t0, $v0
```

```
li $v0, 4  
la $a0, prompt2  
syscall
```

```
li $v0, 5  
syscall  
move $t1, $v0
```

```
add $t2, $t0, $t1
```

```
li $v0, 4  
la $a0, result_add  
syscall
```

```
move $a0, $t2  
li $v0, 1  
syscall
```

```
li $v0, 4  
la $a0, newline  
syscall
```

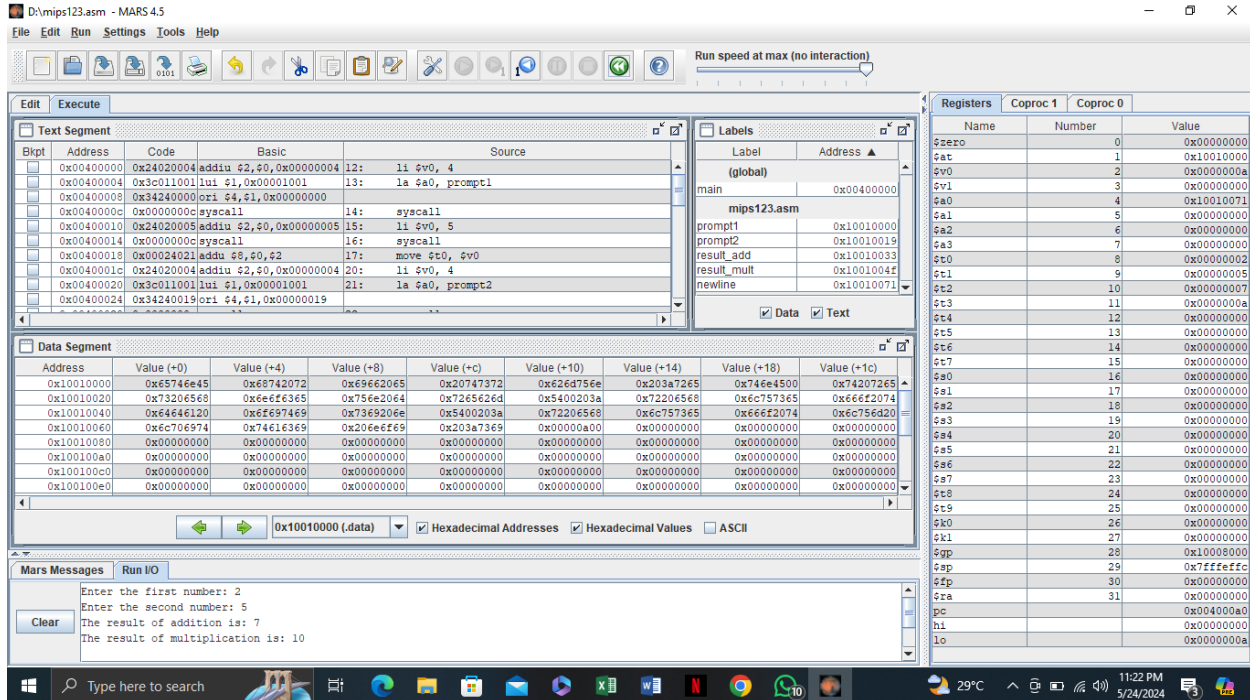
```
mul $t3, $t0, $t1
```

```
li $v0, 4  
la $a0, result_mult  
syscall
```

```
move $a0, $t3  
li $v0, 1  
syscall
```

```
li $v0, 4  
la $a0, newline  
syscall
```

```
li $v0, 10  
syscall
```



Task #1

Write a MIPS program that takes two numbers from the user, divides the first number by the second, and displays the quotient.

```
#Write a MIPS program that takes two numbers from the user,
#divides the first number by the second, and displays the quotient.
```

```
.data
```

```
msg1: .asciiz "Enter 1st number: "
```

```
msg2: .asciiz "Enter 2nd number: "
```

```
msg_result: .asciiz "The quotient is: "
```

```
.text
```

```
li $v0, 4
```

```
la $a0, msg1
```

```
syscall
```



```
li $v0, 5
```

```
syscall
```

```
move $t0, $v0
```

```
li $v0, 4
```

```
la $a0, msg2
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t1, $v0
```

```
div $t0, $t1
```

```
mflo $t2
```

```
li $v0, 4
```

```
la $a0, msg_result
```

```
syscall
```

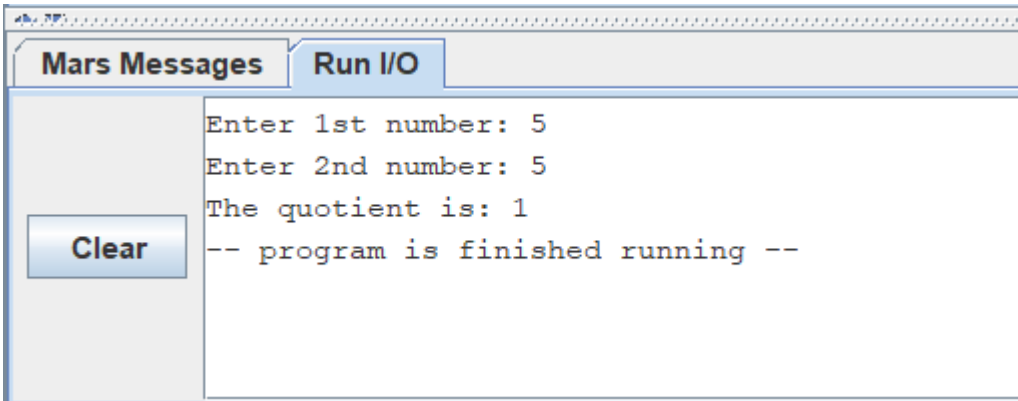
```
li $v0, 1
```

```
move $a0, $t2
```

```
syscall
```

```
li $v0, 10
```

syscall



Task #2

Modify the existing program to perform multiplication of three numbers instead of two. Prompt the user to enter three numbers, multiply them, and display the result of the multiplication.

```
.data
prompt1: .asciiz "Enter the first number: "
prompt2: .asciiz "Enter the second number: "
prompt3: .asciiz "Enter the third number: "
result_mult: .asciiz "The result of multiplication is: "
newline: .asciiz "\n"

.text
    li $v0, 4
    la $a0, prompt1
    syscall

    li $v0, 5
```

```
syscall
```

```
move $t0, $v0
```

```
li $v0, 4
```

```
la $a0, prompt2
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t1, $v0
```

```
li $v0, 4
```

```
la $a0, prompt3
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t2, $v0
```

```
mul $t3, $t0, $t1
```

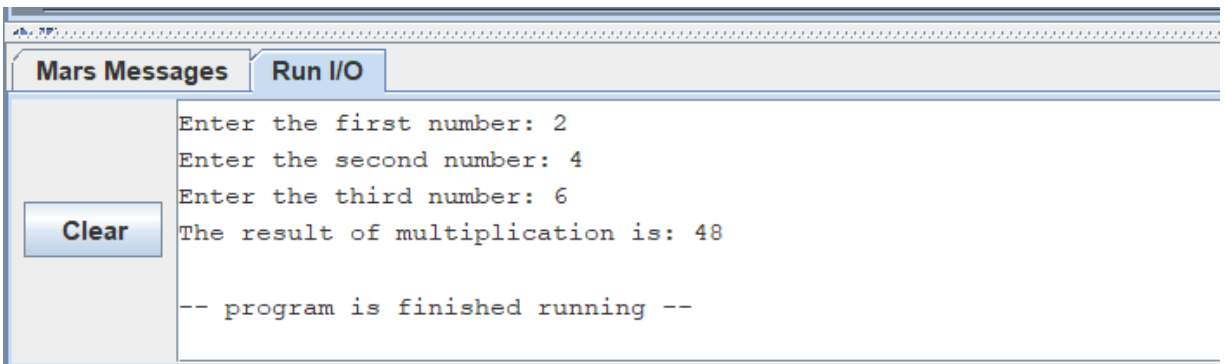
```
mul $t3, $t3, $t2
```

```
li $v0, 4
```

```
la $a0, result_mult
```

```
syscall
```

```
move $a0, $t3  
li $v0, 1  
syscall  
  
li $v0, 4  
la $a0, newline  
syscall  
  
li $v0, 10  
syscall
```



Task #3

Modify the existing program to perform addition of three numbers instead of two. Prompt the user to enter three numbers, add them, and display the result of the addition.

```
#Modify the existing program to perform multiplication of three numbers instead of two.  
  
#Prompt the user to enter three numbers, multiply them, and display the result of the  
multiplication.  
  
.data  
  
prompt1: .asciiz "Enter the first number: "
```

```
prompt2: .asciiz "Enter the second number: "  
prompt3: .asciiz "Enter the third number: "  
result_add: .asciiz "The result of addition is: "  
newline: .asciiz "\n"
```

```
.text
```

```
li $v0, 4
```

```
la $a0, prompt1
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t0, $v0
```

```
li $v0, 4
```

```
la $a0, prompt2
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t1, $v0
```

```
li $v0, 4
```

```
la $a0, prompt3
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t2, $v0
```

```
add $t3, $t0, $t1
```

```
add $t3, $t3, $t2
```

```
li $v0, 4
```

```
la $a0, result_add
```

```
syscall
```

```
move $a0, $t3
```

```
li $v0, 1
```

```
syscall
```

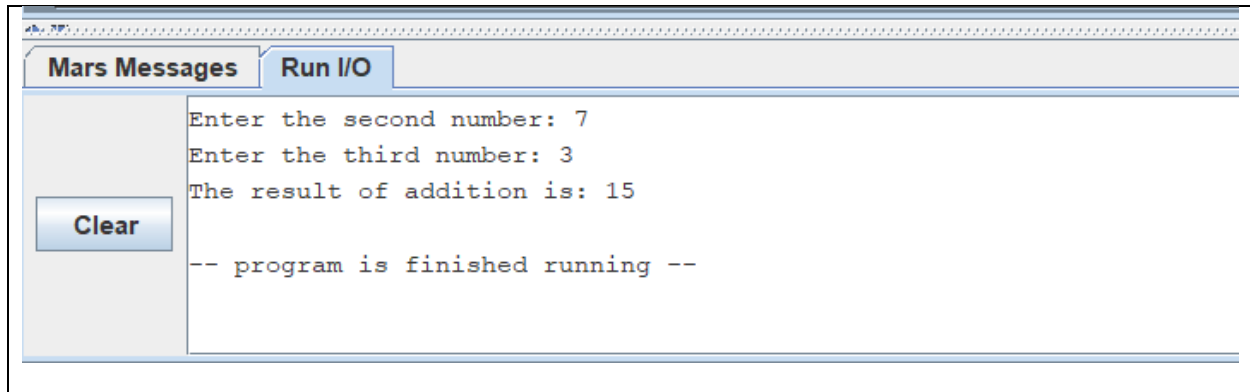
```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```



Discussion and analysis of results:

In this lab session, students performed basic arithmetic operations such as addition, multiplication, subtraction, and division using the MIPS simulator. They analyzed how user input is captured and processed through system calls, and how arithmetic operations are executed using MIPS registers. By testing different input values, students verified the correctness of the results displayed after each operation.

Conclusion:

The lab successfully demonstrated how user input and arithmetic operations can be handled in MIPS assembly language. Students learned to manipulate data in registers and apply system calls to perform input/output operations, reinforcing their understanding of fundamental MIPS programming concepts.

Lab Session 7

Objective:

The objective of this lab is to demonstrate how to store multiple values in an array using the MIPS simulator. Students will learn to define an array in assembly language, manipulate its elements, and access values based on their index positions.

Required Equipment / tools:

- MARS MIPS Simulator

Introduction:

In this lab session, students will explore the concept of arrays in MIPS assembly language. Arrays are essential data structures that allow for the storage and management of multiple values in a single variable. Using the MIPS simulator, students will define an array, store a series of values, and retrieve them through indexing. This exercise will enhance their understanding of memory management, data storage, and how to effectively manipulate arrays in assembly programming. Through practical implementation, students will gain valuable insights into the organization and handling of data in low-level programming.

Procedure:

➤ Array Program:

Input:

```
.data
array: .space 40
newline: .asciiz "\n"
.text
.globl main
main:
    li $t0, 0
    la $t1, array
    li $t2, 10

init_loop:
    li $t3, 2
    sw $t3, 0($t1)
    addi $t1, $t1, 4
    addi $t0, $t0, 1
    blt $t0, $t2, init_loop
```



```
li $t0, 0
la $t1, array
```

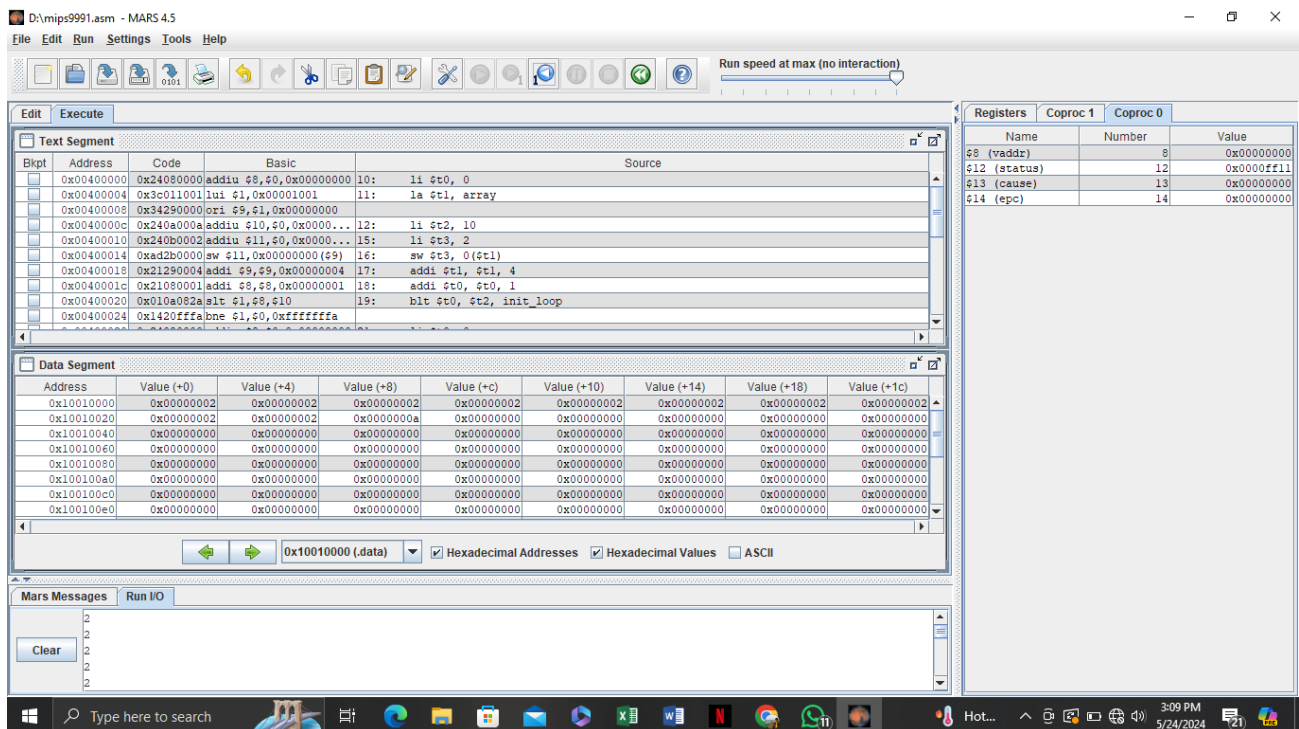
```
print_loop:
lw $a0, 0($t1)
li $v0, 1
syscall
```

```
li $v0, 4
la $a0, newline
syscall
```

```
addi $t1, $t1, 4
addi $t0, $t0, 1
blt $t0, $t2, print_loop
```

```
li $v0, 10
syscall
```

OUTPUT:



Task #1

Modify the existing array program to initialize the array with the first 10 even numbers (0,2,4, ...). Display the values stored in the array after initialization.

```
.data
array:    .space 40
msg:      .asciiz "Even number: \n"

.text

.globl main

main:

    li $t0, 0
    li $t1, 0
    la $t2, array

loop:

    sw $t1, 0($t2)
    addi $t2, $t2, 4
    addi $t1, $t1, 2
    addi $t0, $t0, 1
    blt $t0, 10, loop

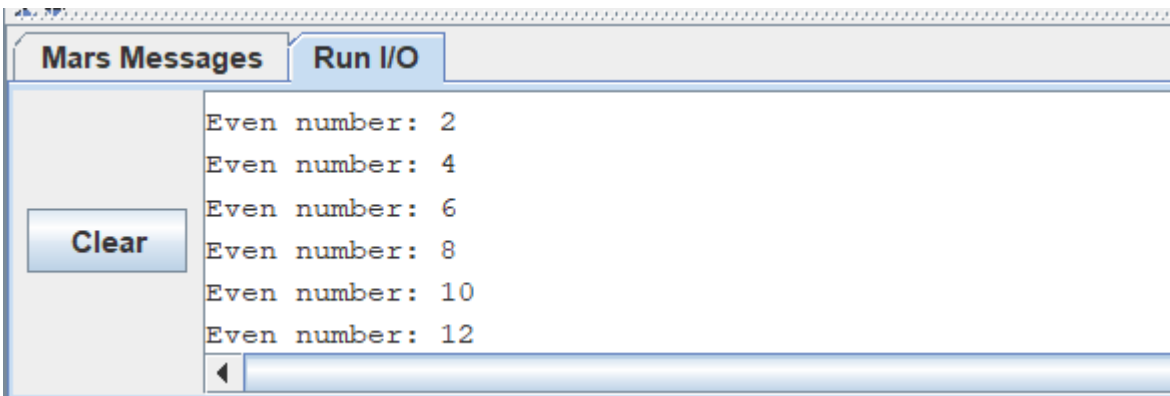
    la $t2, array
    li $t0, 0
```

```
print_loop:
    lw $a0, 0($t2)
    li $v0, 1
    syscall

    la $a0, msg
    li $v0, 4
    syscall

    addi $t2, $t2, 4
    addi $t0, $t0, 1
    blt $t0, 10, print_loop

    li $v0, 10
    syscall
```

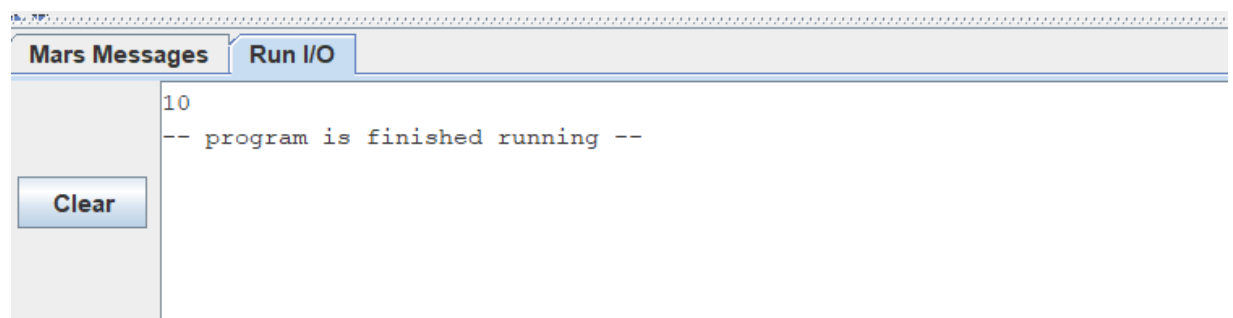


Task #2

Write a MIPS program that retrieves and displays the value at the 5th index of the array. Ensure that the program correctly calculates the index and outputs the value.

```
.data  
  
array: .word 0, 2, 4, 6, 8, 10, 12, 14, 16, 18  
  
.text  
  
.globl main  
  
main:  
  
    la $t0, array  
  
    li $t1, 5  
  
    mul $t1, $t1, 4  
  
    add $t2, $t0, $t1  
  
    lw $t3, 0($t2)  
  
    move $a0, $t3  
  
    li $v0, 1  
  
    syscall  
  
    li $v0, 10  
  
    syscall
```

4 array: *word* 0, 2, 4, 6, 8, 10, 12, 14, 16, 18



Task #3

Create a MIPS program that allows the user to input a value and store it in the 3rd index of the array. After storing the value, display the entire array to confirm the modification.

```
.data
array: .space 40
prompt: .asciiz "Enter a value: "
newline: .asciiz "\n"

.text
.globl main

main:
    la $t0, array
    li $t1, 0

init_loop:
    sw $t1, 0($t0)
    addi $t0, $t0, 4
    addi $t1, $t1, 1
    blt $t1, 10, init_loop

    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
```

```
syscall
```

```
la $t0, array
```

```
li $t1, 3
```

```
mul $t1, $t1, 4
```

```
add $t2, $t0, $t1
```

```
sw $v0, 0($t2)
```

```
la $t0, array
```

```
li $t1, 0
```

```
display_loop:
```

```
lw $a0, 0($t0)
```

```
li $v0, 1
```

```
syscall
```

```
la $a0, newline
```

```
li $v0, 4
```

```
syscall
```

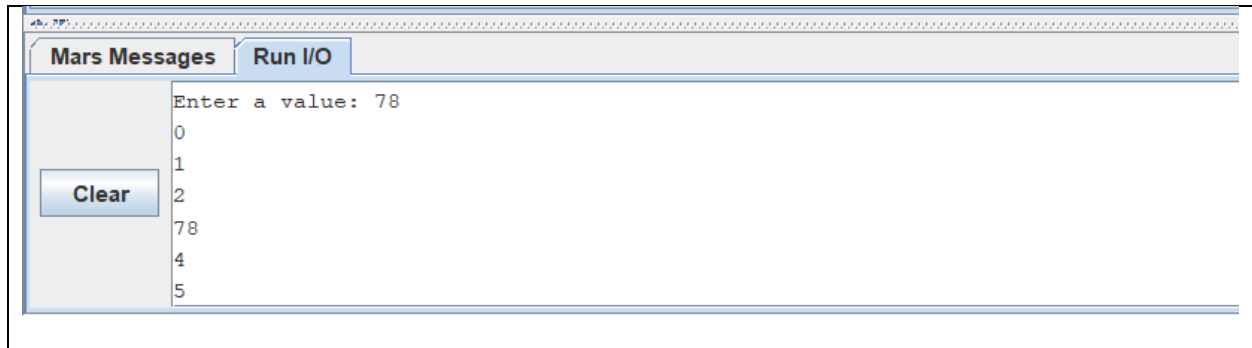
```
addi $t0, $t0, 4
```

```
addi $t1, $t1, 1
```

```
blt $t1, 10, display_loop
```

```
li $v0, 10
```

```
syscall
```



Discussion and analysis of results:

In this section, students will discuss and analyze the results of their practical tasks involving arrays in MIPS assembly language. They will evaluate how the array's initialization, retrieval, and modification were implemented and the impact of these operations on memory management. By observing the behavior of their programs, students can gain insights into how data is organized and accessed in low-level programming, enhancing their understanding of array manipulation and its applications.

Conclusion:

The lab session provided hands-on experience in working with arrays in MIPS assembly language, reinforcing the importance of data structures in programming. Students learned how to efficiently store, retrieve, and modify multiple values in memory. This understanding is crucial for developing more complex algorithms and programs that require effective data management and manipulation in assembly language. Overall, the lab emphasized the foundational role of arrays in computer science and their relevance in efficient programming practices.

Lab Session 8

Open Ended Lab 1

CLOs	Mapped GAs	Bloom Taxonomy
CLO 3	GA-4 Design/Development of Solutions	C6 (Creating)

Motivation

Background/Theory:

Procedure/Methodology:

Task 1:

Objective

Task 2:

Objective:

Lab Session 9

Objective:

The objective of this lab is to perform operations on floating-point and double values using the MIPS simulator. Students will learn to define, manipulate, and compute with floating-point numbers and double-precision values.

Required Equipment / tools:

- MARS MIPS Simulator

Introduction:

In this lab session, students will explore the handling of floating-point and double values in MIPS assembly language. They will learn to define these data types and perform arithmetic operations, understanding their importance in representing real numbers. Using the MIPS simulator, students will gain hands-on experience with the specific instructions required for floating-point calculations, enhancing their knowledge of numerical precision and memory representation in low-level programming.

Procedure:

➤ Floating Point & Multiplying Double Values & Adding Double Values Program:

INPUT :

```
.data
prompt_float1: .asciiz "Enter the first floating-point number: "
prompt_float2: .asciiz "Enter the second floating-point number: "
result_add: .asciiz "The result of addition is: "
result_mult: .asciiz "The result of multiplication is: "
newline: .asciiz "\n"
.text
.globl main
main:
    li $v0, 4
    la $a0, prompt_float1
    syscall
    li $v0, 6
    syscall
    li $v0, 4
    la $a0, prompt_float2
    syscall
```

```

    li $v0, 6
    syscall
    add.s $f4, $f0, $f2
    li $v0, 4
    la $a0, result_add
    syscall
    mov.s $f12, $f4
    li $v0, 2
    syscall
    li $v0, 4
    la $a0, newline
    syscall
    mul.s $f6, $f0, $f2
    li $v0, 4
    la $a0, result_mult
    syscall
    mov.s $f12, $f6
    li $v0, 2
    syscall
    li $v0, 4
    la $a0, newline
    syscall
    li $v0, 10
    syscall

```

OUTPUT:

The screenshot shows the MARS 4.5 MIPS simulator interface. The main window displays assembly code with columns for Address, Code, Basic, and Source. The registers window on the right shows the state of various registers, including \$zero, \$at, \$v0, \$a0, \$a1, \$a2, \$a3, \$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7, \$a0, \$a1, \$a2, \$a3, \$a4, \$a5, \$a6, \$a7, \$a8, \$a9, \$a10, \$a11, \$a12, \$a13, \$a14, \$a15, \$a16, \$a17, \$a18, \$a19, \$a20, \$a21, \$a22, \$a23, \$a24, \$a25, \$a26, \$a27, \$a28, \$a29, \$a30, \$a31, \$pc, \$hi, and \$lo. The console window at the bottom shows the execution results, including the first floating-point number (4), the second floating-point number (2), the result of addition (2.0), and the result of multiplication (0.0).

Task #1

Modify the existing program to perform addition of three floating-point numbers instead of two. Prompt the user for three values and display the result of their addition.

```
.data

prompt1: .asciiz "Enter the first number: "
prompt2: .asciiz "Enter the second number: "
prompt3: .asciiz "Enter the third number: "
resultMsg: .asciiz "The result of the addition is: "

.text

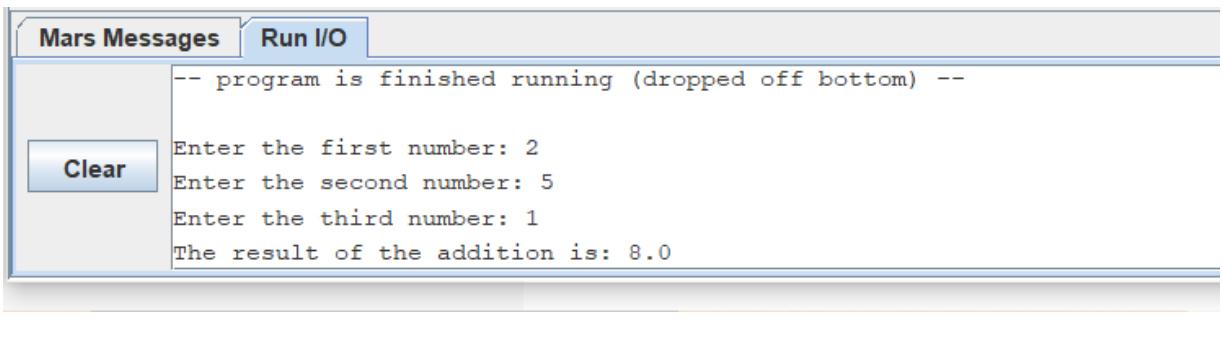
li $v0, 4
la $a0, prompt1
syscall

li $v0, 6
syscall
mov.s $f2, $f0

li $v0, 4
la $a0, prompt2
syscall

li $v0, 6
syscall
mov.s $f4, $f0
```

```
li $v0, 4  
la $a0, prompt3  
syscall  
  
li $v0, 6  
syscall  
mov.s $f6, $f0  
  
add.s $f12, $f2, $f4  
add.s $f12, $f12, $f6  
  
li $v0, 4  
la $a0, resultMsg  
syscall  
  
li $v0, 2  
syscall
```



Task #2

Write a MIPS program that takes two floating-point numbers as input from the user, divides the first number by the second, and displays the result. Ensure to handle cases where the second number is zero to avoid division by zero errors.

```
.data

prompt1: .asciiz "Enter the first number: "
prompt2: .asciiz "Enter the second number: "
resultMsg: .asciiz "The result of the division is: "
errorMsg: .asciiz "Error: Division by zero is not allowed.\n"

.text

li $v0, 4
la $a0, prompt1
syscall

li $v0, 6
syscall
mov.s $f2, $f0

li $v0, 4
la $a0, prompt2
syscall

li $v0, 6
syscall
mov.s $f4, $f0
```

```
c.eq.s $f4, $f0
```

```
bc1f no_zero
```

```
li $v0, 4
```

```
la $a0, errorMsg
```

```
syscall
```

```
j end
```

```
no_zero:
```

```
div.s $f12, $f2, $f4
```

```
li $v0, 4
```

```
la $a0, resultMsg
```

```
syscall
```

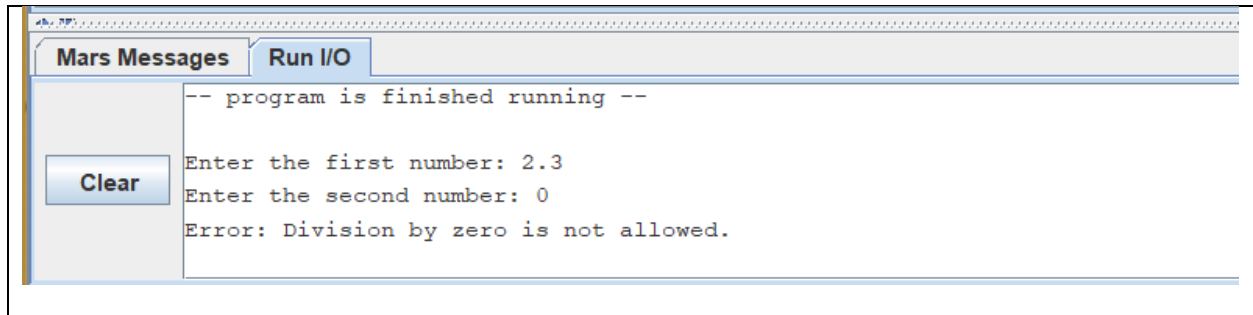
```
li $v0, 2
```

```
syscall
```

```
end:
```

```
li $v0, 10
```

```
syscall
```



Task #3

Create a MIPS program that performs addition and multiplication using double-precision values. Prompt the user to enter two double values and display both the sum and the product.

```
.data  
  
prompt1: .asciiz "Enter the first double value: "  
prompt2: .asciiz "Enter the second double value: "  
resultMsg1: .asciiz "The sum of the values is: "  
resultMsg2: .asciiz "The product of the values is: "  
  
.text  
  
li $v0, 4  
la $a0, prompt1  
syscall  
  
li $v0, 6  
syscall  
mov.d $f2, $f0  
  
li $v0, 4  
la $a0, prompt2
```

syscall

li \$v0, 6

syscall

mov.d \$f4, \$f0

add.d \$f12, \$f2, \$f4

li \$v0, 4

la \$a0, resultMsg1

syscall

li \$v0, 3

syscall

mul.d \$f12, \$f2, \$f4

li \$v0, 4

la \$a0, resultMsg2

syscall

li \$v0, 3

syscall

li \$v0, 10

syscall

Discussion and analysis of results:

In this section, students will discuss and analyze the results obtained from performing operations on floating-point and double values. They will evaluate the accuracy and precision of the arithmetic operations, considering how the representation of real numbers in MIPS affects calculations. By examining their implementation, students can identify any discrepancies or unexpected outcomes, deepening their understanding of floating-point arithmetic.

Conclusion:

This lab session provided valuable insights into working with floating-point and double values in MIPS assembly language. Students learned the importance of precision and proper handling of these data types during arithmetic operations. Overall, the practical exercises reinforced theoretical concepts and highlighted the challenges and considerations in numerical computing within low-level programming.

Lab Session 10

Objective:

The objective of this lab is to perform comparisons between integer and floating-point values using the MIPS simulator. Students will gain practical experience in using MIPS instructions for evaluating and interpreting the results of these comparisons.

Required Equipment / tools:

- MARS MIPS Simulator

Introduction:

In this lab session, students will examine how to compare different data types, specifically integers and floating-point numbers, in MIPS assembly language. They will learn to utilize MIPS comparison instructions and understand how these operations influence program flow through conditional branching. By practicing these comparisons, students will develop a deeper understanding of how to implement decision-making logic in assembly programming. This hands-on experience is essential for mastering control structures and refining their skills in low-level programming.

Procedure:

➤ Number & FLOAT Comparison:

INPUT:

```
.data
prompt_int1: .asciiz "Enter the first integer: "
prompt_int2: .asciiz "Enter the second integer: "
prompt_float1: .asciiz "Enter the first floating-point number: "
prompt_float2: .asciiz "Enter the second floating-point number: "
int_equal: .asciiz "The integers are equal.\n"
int_not_equal: .asciiz "The integers are not equal.\n"
float_equal: .asciiz "The floating-point numbers are equal.\n"
float_not_equal: .asciiz "The floating-point numbers are not equal.\n"
```

```
.text
main:
```

```
    li $v0, 4
    la $a0, prompt_int1
    syscall
    li $v0, 5
    syscall
```

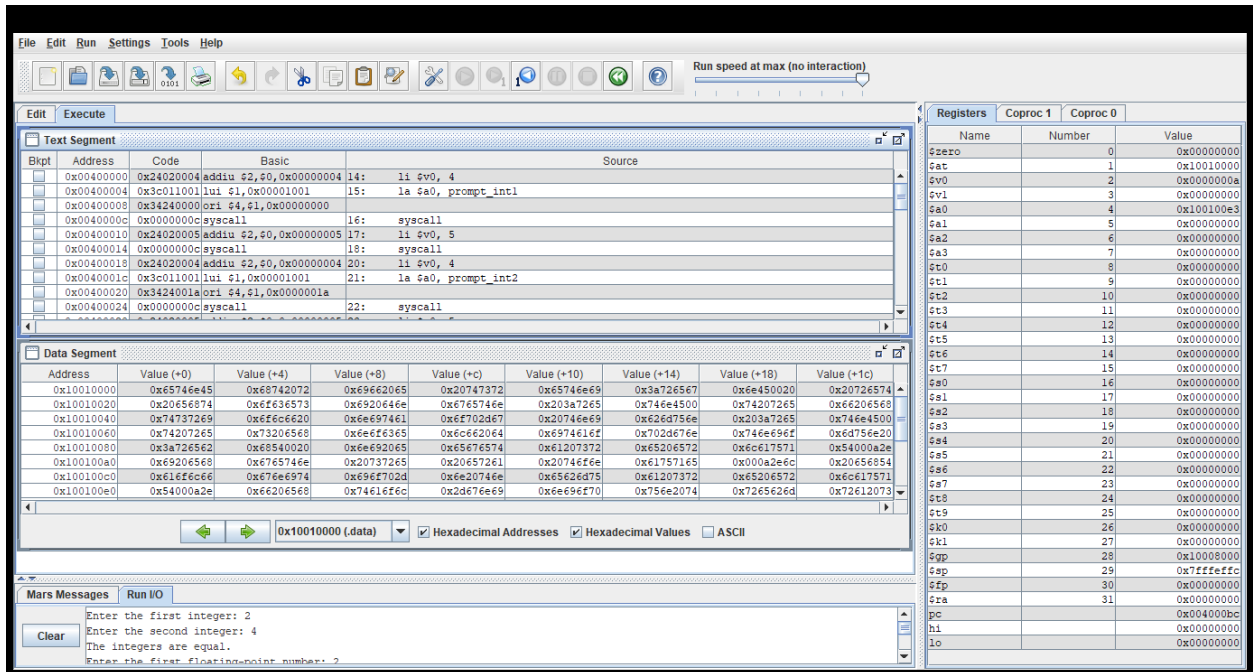
```
li $v0, 4
la $a0, prompt_int2
syscall
li $v0, 5
syscall
beq $t0, $t1, integers_equal
li $v0, 4
la $a0, int_not_equal
syscall
j compare_floats

integers_equal:
li $v0, 4
la $a0, int_equal
syscall

compare_floats:

li $v0, 4
la $a0, prompt_float1
syscall
li $v0, 6
syscall
li $v0, 4
la $a0, prompt_float2
syscall
li $v0, 6
syscall
c.eq.s $f0, $f2
bc1t floats_equal
li $v0, 4
la $a0, float_not_equal
syscall
j end
floats_equal:
li $v0, 4
la $a0, float_equal
syscall
end:
li $v0, 10
syscall
```

OUTPUT:



Task #1

Modify the existing program to include a comparison that checks whether the first integer is greater than or less than the second integer, and display appropriate messages for each case.

.data

prompt1: .asciiz "Enter the first integer: "

prompt2: .asciiz "Enter the second integer: "

resultMsg1: .asciiz "The sum of the values is: "

resultMsg2: .asciiz "The product of the values is: "

greaterMsg: .asciiz "The first integer is greater than the second integer.\n"

lessMsg: .asciiz "The first integer is less than the second integer.\n"

equalMsg: .asciiz "The first integer is equal to the second integer.\n"

.text

li \$v0, 4

la \$a0, prompt1

```
syscall

li $v0, 5
syscall
move $t0, $v0      # Store first integer in $t0

li $v0, 4
la $a0, prompt2
syscall

li $v0, 5
syscall
move $t1, $v0      # Store second integer in $t1

# Comparison
bgt $t0, $t1, greater
blt $t0, $t1, less
beq $t0, $t1, equal

greater:
li $v0, 4
la $a0, greaterMsg
syscall
j end
```

less:

```
li $v0, 4
```

```
la $a0, lessMsg
```

```
syscall
```

```
j end
```

equal:

```
li $v0, 4
```

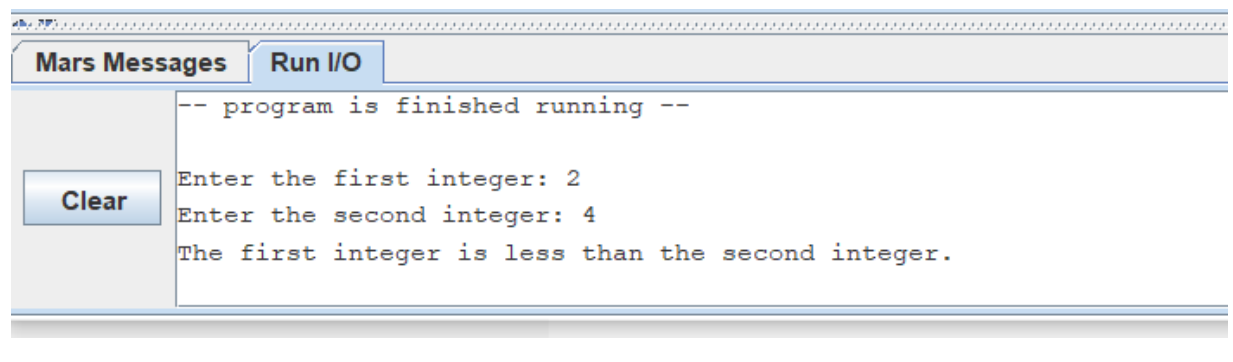
```
la $a0, equalMsg
```

```
syscall
```

end:

```
li $v0, 10
```

```
syscall
```



Task #2

Create a MIPS program that takes one integer and one floating-point number as input, compares them for equality, and displays whether they are equal. Make sure to cast the integer to a float for accurate comparison before evaluating.

```
.data

prompt1: .asciiz "Enter an integer: "
prompt2: .asciiz "Enter a floating-point number: "
equalMsg: .asciiz "The values are equal.\n"
notEqualMsg: .asciiz "The values are not equal.\n"


.text

li $v0, 4
la $a0, prompt1
syscall


li $v0, 5
syscall
move $t0, $v0      # Store integer in $t0


li $v0, 4
la $a0, prompt2
syscall


li $v0, 6
syscall
mov.s $f2, $f0      # Store floating-point number in $f2


# Cast the integer to float by moving it into a floating-point register
mtc1 $t0, $f4        # Move integer from $t0 to floating-point register $f4
```

```
cvt.s.w $f4, $f4    # Convert the integer in $f4 to float

# Compare the integer (cast to float) and the floating-point number

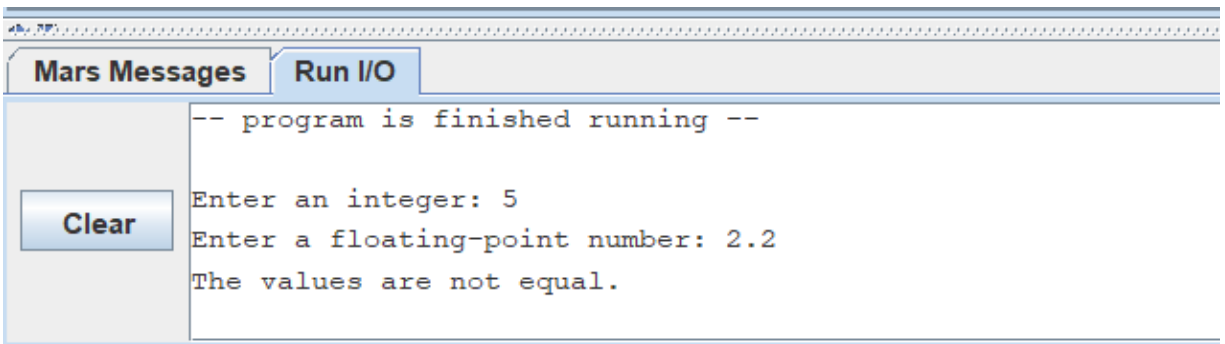
c.eq.s $f4, $f2     # Compare $f4 (casted integer) with $f2 (float)

bc1t equal          # If equal, branch to equal

li $v0, 4
la $a0, notEqualMsg
syscall
j end

equal:
li $v0, 4
la $a0, equalMsg
syscall

end:
li $v0, 10
syscall
```



Task #3

Modify the existing program to display a message indicating which data type (integer or floating-point) is greater if they are not equal. For example, if the integer is greater than the floating-point number, display "The integer is greater," and vice versa.

```
.data

prompt1: .asciiz "Enter an integer: "
prompt2: .asciiz "Enter a floating-point number: "
equalMsg: .asciiz "The values are equal.\n"
notEqualMsg: .asciiz "The values are not equal.\n"
intGreaterMsg: .asciiz "The integer is greater.\n"
floatGreaterMsg: .asciiz "The floating-point number is greater.\n"

.text

li $v0, 4
la $a0, prompt1
syscall

li $v0, 5
syscall
move $t0, $v0      # Store integer in $t0

li $v0, 4
la $a0, prompt2
syscall
```

```
li $v0, 6

syscall

mov.s $f2, $f0    # Store floating-point number in $f2

# Cast the integer to float by moving it into a floating-point register
mtc1 $t0, $f4     # Move integer from $t0 to floating-point register $f4
cvt.s.w $f4, $f4   # Convert the integer in $f4 to float

# Compare the integer (casted to float) and the floating-point number
c.eq.s $f4, $f2    # Compare $f4 (casted integer) with $f2 (float)
bc1t equal         # If equal, branch to equal

# If they are not equal, check which one is greater
c.lt.s $f4, $f2    # Check if integer (casted) is less than floating-point number
bc1t floatGreater  # If true, branch to floatGreater

li $v0, 4

la $a0, intGreaterMsg

syscall

j end

floatGreater:

li $v0, 4

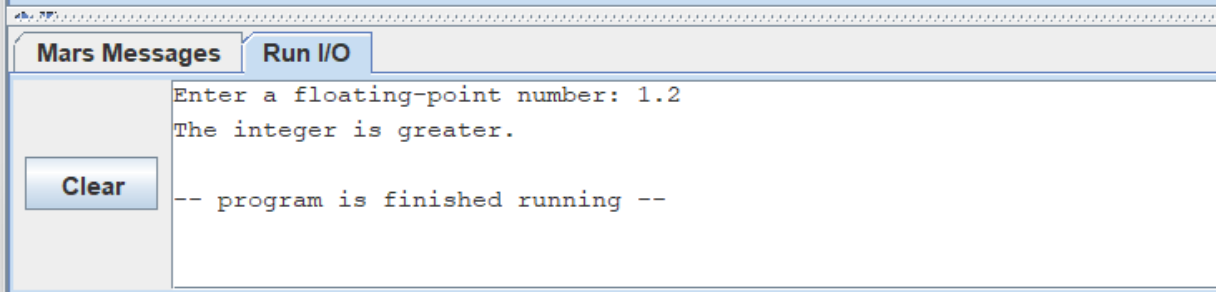
la $a0, floatGreaterMsg

syscall
```

```
j end

equal:
    li $v0, 4
    la $a0, equalMsg
    syscall
    j end

end:
    li $v0, 10
    syscall
```



Discussion and analysis of results:

In this lab session, students successfully implemented comparisons between integer and floating-point values using the MIPS simulator. The results demonstrated an understanding of how MIPS instructions facilitate conditional branching based on the outcomes of these comparisons. Observations highlighted the significance of data type handling and its impact on decision-making in assembly programming.

Conclusion:

The lab effectively reinforced the concepts of data comparison in MIPS assembly language. Students gained practical experience in utilizing comparison instructions and interpreting results, which are essential skills for developing efficient algorithms in low-level programming. This hands-on approach deepened their understanding of control structures and enhanced their ability to manage data types effectively.

Lab Session 11

Objective:

To understand and implement the concept of loops in MIPS assembly language using the while loop structure. To develop skills in using the MIPS simulator for executing and debugging programs that utilize control flow mechanisms

Required Equipment / tools:

- MARS MIPS Simulator

Introduction:

In this lab session, students will explore the implementation of while loops in MIPS assembly language, a fundamental programming structure that enables repetitive execution of code blocks based on conditional evaluations. Through practical exercises in the MIPS simulator, participants will gain hands-on experience in structuring loops, managing control flow, and effectively debugging assembly code. This session aims to enhance understanding of program logic and improve proficiency in MIPS programming techniques.

Procedure:

➤ While Loop Program:

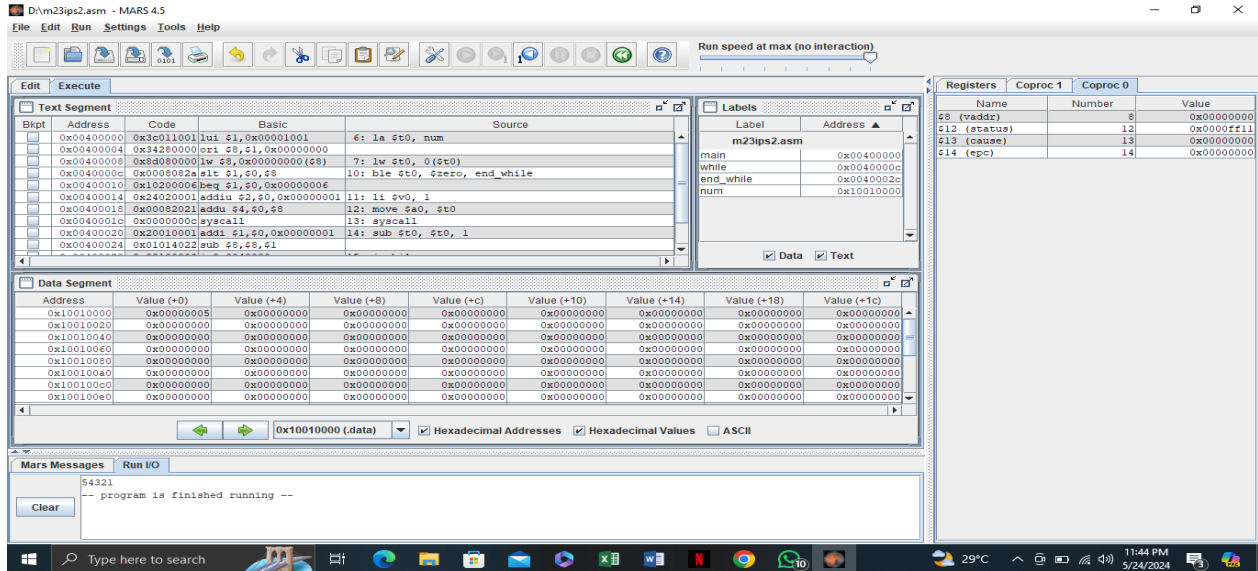
```
.data
num: .word 5

.text
main:
la $t0, num
lw $t0, 0($t0)

while:
ble $t0, $zero, end_while
li $v0, 1
move $a0, $t0
syscall
sub $t0, $t0, 1
j while
```

```
end_while:
li $v0, 10
syscall
```

OUTPUT:



Task #1

Modify the existing while loop program to decrement the value of num by 2 instead of 1. What is the output when you run the modified program?

.data

num: .word 10

.text

.globl main

main:

lw \$t0, num

li \$v0, 1

loop:

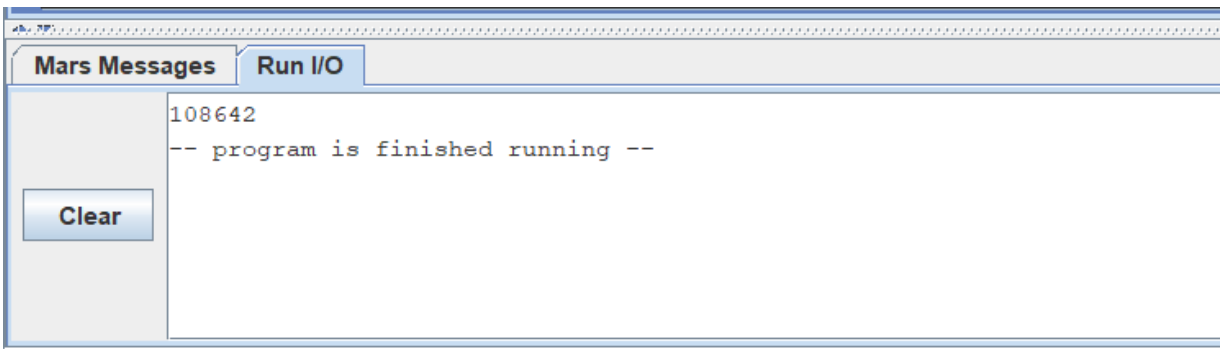
blez \$t0, exit

move \$a0, \$t0

```
syscall  
  
subi $t0, $t0, 2  
  
j loop
```

exit:

```
li $v0, 10  
  
syscall
```



Task #2

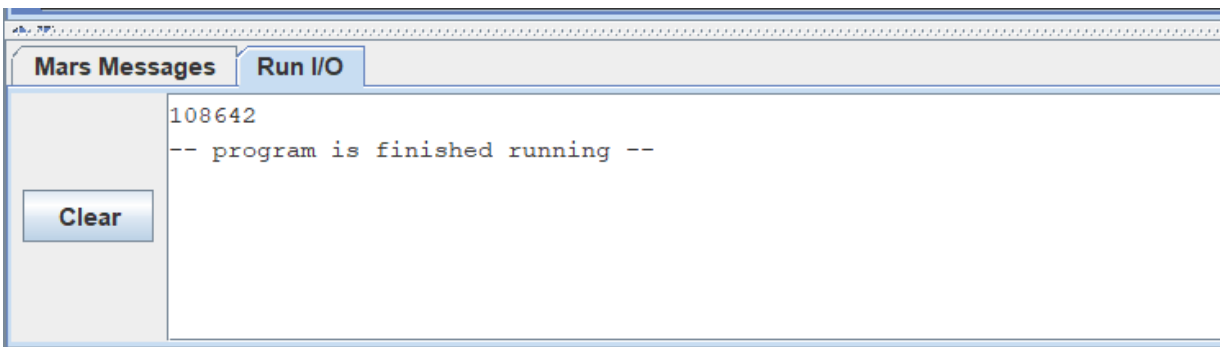
Change the initial value of num to 10 in the program. What will be the output when you execute the updated code?

```
.data  
  
num: .word 10  
  
newline: .asciiz "\n"  
  
.text  
  
.globl main  
  
main:
```

```
lw $t0, num
li $v0, 1

loop:
    blez $t0, exit
    move $a0, $t0
    syscall
    la $a0, newline
    li $v0, 4
    syscall
    subi $t0, $t0, 2
    j loop

exit:
    li $v0, 10
    syscall
```



Task #3

Modify the program to print only even numbers from the initial value of num down to zero.
What adjustments did you make to achieve this output?

```
.data

num: .word 10


.text

.globl main

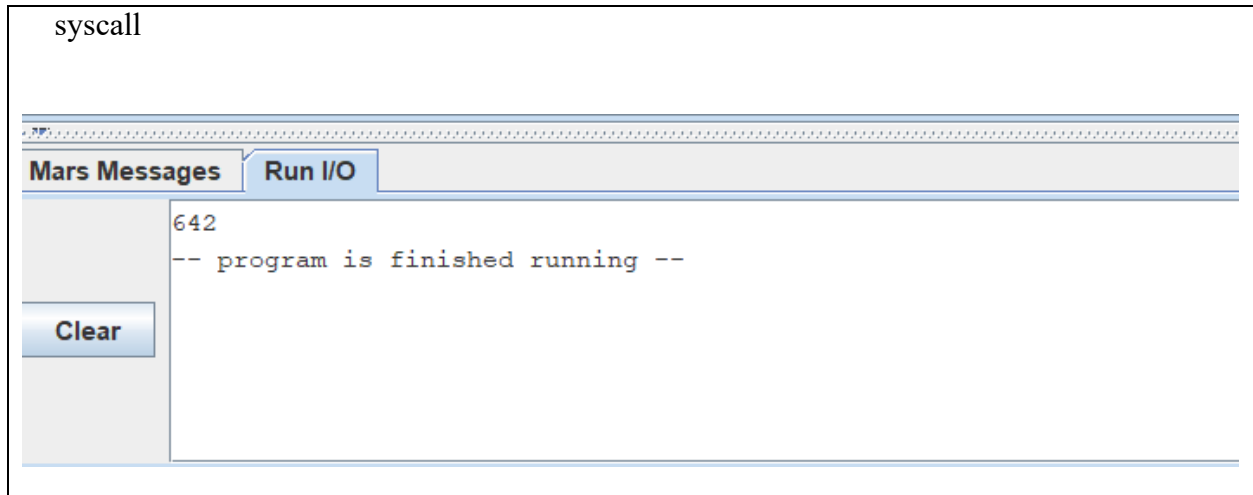
main:

    lw $t0, num
    li $v0, 1

loop:
    blez $t0, exit
    andi $t1, $t0, 1
    beqz $t1, print_num
    subi $t0, $t0, 2
    j loop

print_num:
    move $a0, $t0
    syscall
    subi $t0, $t0, 2
    j loop

exit:
    li $v0, 10
```

Discussion and analysis of results:

In this section, we will evaluate the output generated by the modified MIPS assembly programs using while loops. We will analyze how the changes made to the code, such as altering the decrement value, adjusting the initial number, or adding custom messages, impacted the overall functionality and output. Additionally, we will discuss any challenges encountered during the execution and debugging process, and how these experiences contributed to a deeper understanding of control flow in MIPS assembly language.

Conclusion:

The practical implementation of while loops in MIPS assembly language provided valuable insights into the structure and control flow of assembly programming. Through the modifications made to the original program, students gained hands-on experience in programming techniques and debugging processes. Overall, this lab session enhanced their proficiency in MIPS programming and reinforced the importance of iterative constructs in software development.

Lab Session 12

Objective:

To implement a program in MIPS assembly language that calculates the factorial of a user-defined number. To enhance understanding of control structures, loops, and user input handling in MIPS assembly programming.

Required Equipment / tools:

- MARS MIPS Simulator

Introduction:

In this lab session, we will explore the concept of calculating the factorial of a number using the MIPS simulator. The factorial of a non-negative integer nnn is the product of all positive integers less than or equal to nnn and is denoted as $n!n!n!$. By utilizing MIPS assembly language, students will gain practical experience in manipulating data and implementing iterative algorithms, thereby reinforcing their knowledge of fundamental programming constructs in assembly language. This session aims to deepen the understanding of arithmetic operations and control flow in a low-level programming environment.

Procedure:

➤ Factorial Program:

INPUT:

```
.data
prompt: .asciiz "Enter a number: "
result: .asciiz "Factorial: "
```

```
.text
main:
li $v0, 4
la $a0, prompt
syscall
```

```
li $v0, 5
syscall
move $t0, $v0
```

```
li $t1, 1
factorial:
ble $t0, $zero, end_factorial
mul $t1, $t1, $t0
```

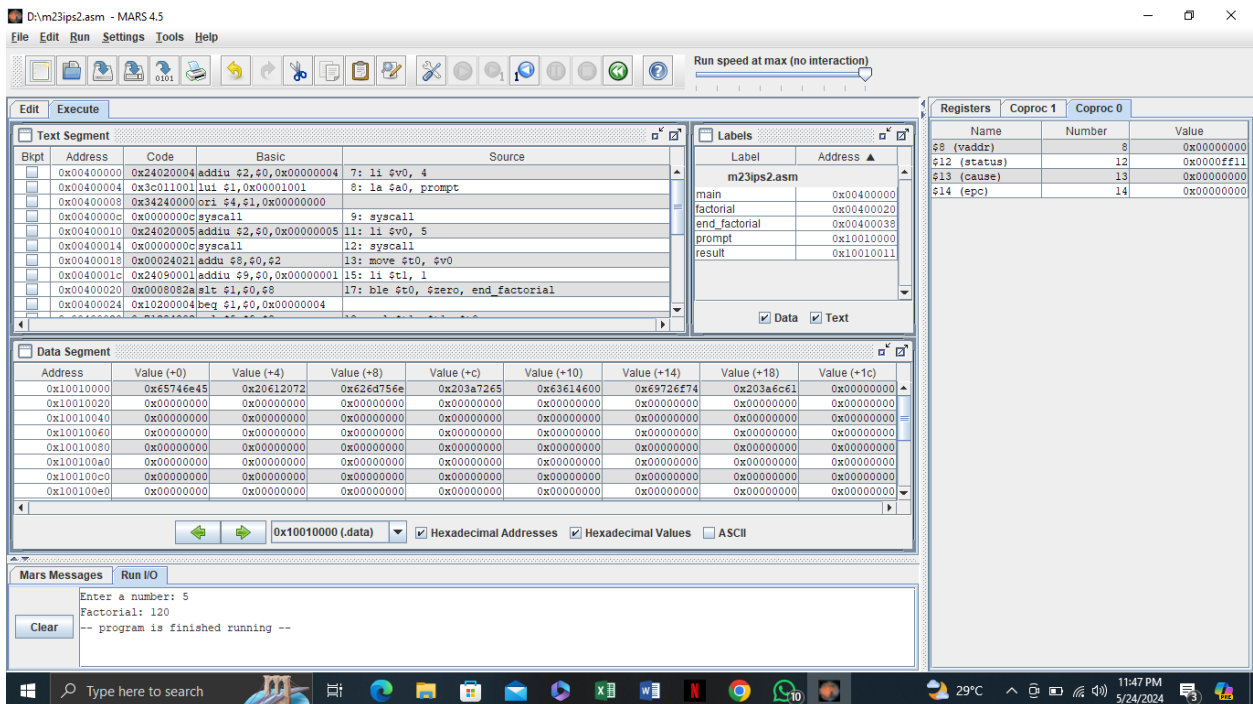
```
sub $t0, $t0, 1
j factorial
```

```
end_factorial:
li $v0, 4
la $a0, result
syscall
```

```
li $v0, 1
move $a0, $t1
syscall
```

```
li $v0, 10
syscall
```

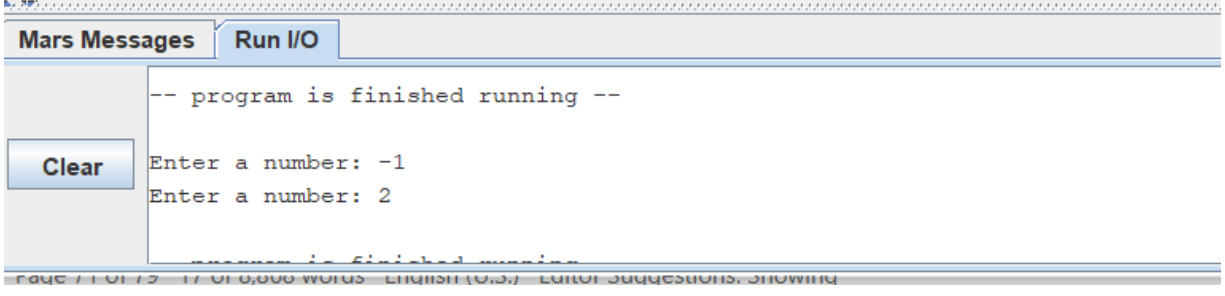
OUTPUT:



Task #1

Add a check to handle cases where the user enters a negative number, prompting a suitable message.

S



```
.data
prompt: .asciiz "Enter a number: "
negativeMsg: .asciiz "The number is negative.\n"
.text
.globl main

main:

    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall

    move $t0, $v0

    # Check if the number is negative
```

```
bltz $t0, negative  
  
# Exit program if number is not negative  
  
li $v0, 10  
  
syscall  
  
negative:  
  
li $v0, 4  
  
la $a0, negativeMsg  
  
syscall  
  
li $v0, 10  
  
syscall
```

Task #2

Modify the program to explicitly handle the case where the user inputs 0, returning 1 as the factorial.

```
.data  
  
prompt: .asciiz "Enter a number: "  
negativeMsg: .asciiz "The number is negative.\n"  
factorialMsg: .asciiz "The factorial is: "  
newline: .asciiz "\n"  
  
.text  
  
.globl main  
  
main:  
  
li $v0, 4  
  
la $a0, prompt
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t0, $v0
```

```
bltz $t0, negative
```

```
beqz $t0, zero_case
```

```
li $t1, 1
```

```
li $t2, 1
```

```
factorial_loop:
```

```
mul $t1, $t1, $t2
```

```
addi $t2, $t2, 1
```

```
bgt $t2, $t0, print_factorial
```

```
j factorial_loop
```

```
zero_case:
```

```
li $t1, 1
```

```
j print_factorial
```

```
print_factorial:
```

```
li $v0, 4
```

```
la $a0, factorialMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t1
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```

```
negative:
```

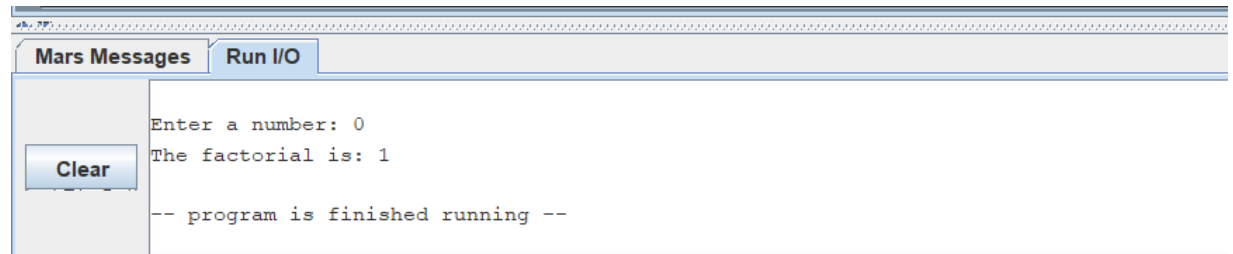
```
li $v0, 4
```

```
la $a0, negativeMsg
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```



Task #3

Enhance the program to count and display the number of multiplication steps performed in the factorial calculation.

Expected Output:

For input 4, the output should include:

"Factorial: 24, Steps: 4"

```
.data
prompt: .asciiz "Enter a number: "
negativeMsg: .asciiz "The number is negative.\n"
factorialMsg: .asciiz "Factorial: "
stepsMsg: .asciiz "Steps: "
newline: .asciiz "\n"
.text
.globl main

main:
    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    move $t0, $v0

    bltz $t0, negative
```



```
beqz $t0, zero_case
```

```
li $t1, 1
```

```
li $t2, 1
```

```
li $t3, 0
```

```
factorial_loop:
```

```
mul $t1, $t1, $t2
```

```
addi $t2, $t2, 1
```

```
addi $t3, $t3, 1
```

```
bgt $t2, $t0, print_result
```

```
j factorial_loop
```

```
zero_case:
```

```
li $t1, 1
```

```
li $t3, 0
```

```
j print_result
```

```
print_result:
```

```
li $v0, 4
```

```
la $a0, factorialMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t1
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, stepsMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t3
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```

```
negative:
```

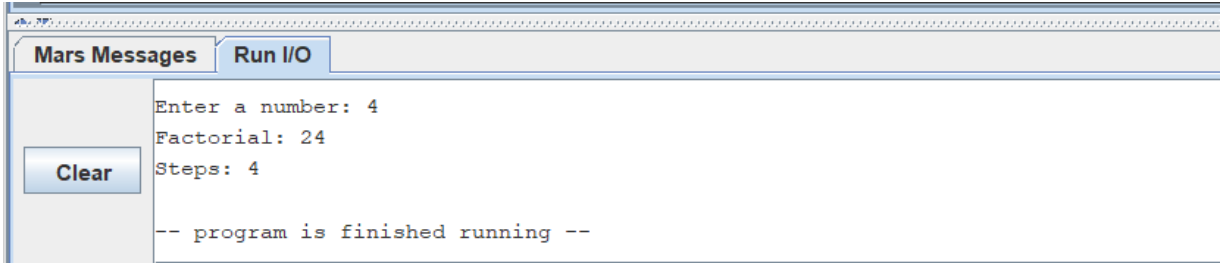
```
li $v0, 4
```

```
la $a0, negativeMsg
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```



Discussion and analysis of results:

In this section, we will evaluate the performance of the modified MIPS assembly program for calculating the factorial of a user-defined number. The various enhancements, such as input validation, counting calculation steps, and handling special cases like zero and negative numbers, contribute to a more robust and user-friendly application. Analyzing the outputs obtained from different test cases will highlight the effectiveness of these modifications and their impact on the program's reliability and usability.

Conclusion:

The lab session successfully demonstrated the implementation of a factorial calculation program using MIPS assembly language. Through the practical tasks, students gained valuable insights into control structures, loops, and user input handling, which are fundamental to assembly programming. The enhancements made to the program not only improved its functionality but also reinforced the importance of validating user input and managing program flow effectively. Overall, this exercise deepened students' understanding of low-level programming concepts and their applications in real-world scenarios.

Lab Session 13

Objective:

To implement a program in MIPS assembly language that calculates the average of a set of user-defined numbers. To enhance understanding of data handling, loops, and arithmetic operations in MIPS assembly programming.

Required Equipment / tools:

MARS MIPS Simulator

Introduction:

In this lab session, we will focus on calculating the average of a series of numbers using the MIPS simulator. The average is computed by summing a given set of numbers and dividing the total by the count of those numbers. Through this exercise, students will gain hands-on experience in managing user input, implementing control structures, and performing arithmetic operations in MIPS assembly language. This practical task aims to strengthen their understanding of programming concepts in a low-level environment, reinforcing essential skills in data manipulation and algorithm design.

Procedure:

➤ Average Program:

INPUT :

```
.data
prompt: .asciiz "Enter a number (0 to finish): "
result: .asciiz "Average: "
```

```
.text
main:
li $t0, 0
li $t1, 0
li $t2, 0
```

```
loop:
li $v0, 4
la $a0, prompt
syscall
```

```
li $v0, 5
syscall
```

move \$t3, \$v0

beq \$t3, \$zero, end_loop

add \$t0, \$t0, \$t3

add \$t1, \$t1, 1

j loop

end_loop:

div \$t2, \$t0, \$t1

li \$v0, 4

la \$a0, result

syscall

li \$v0, 2

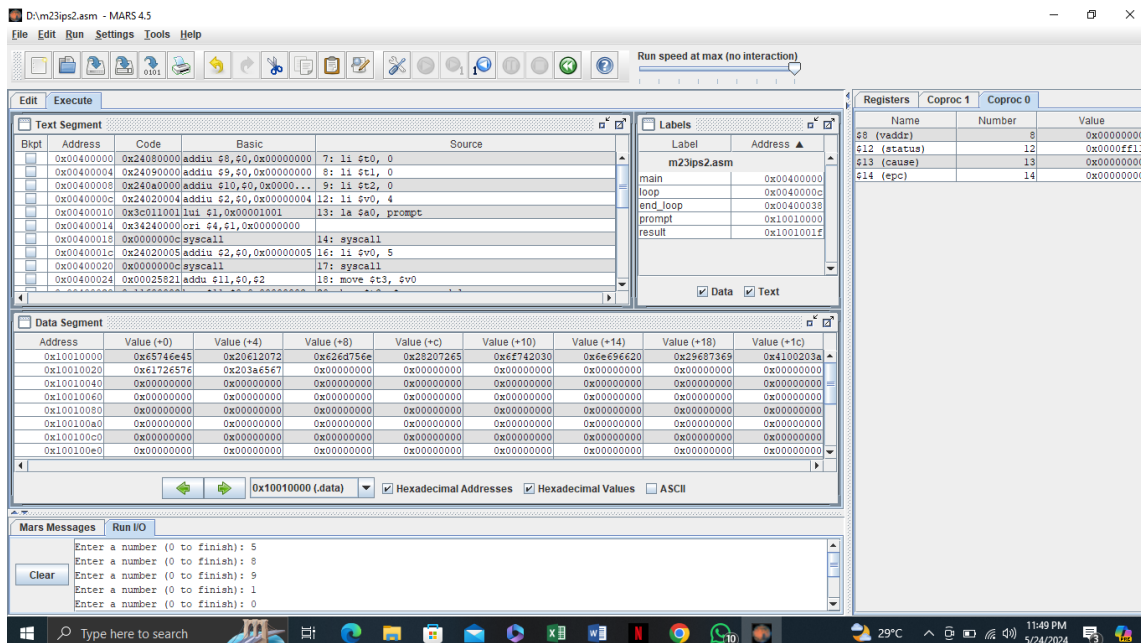
move \$a0, \$t2

syscall

li \$v0, 10

syscall

OUTPUT:



Task #1

Enhance the program to display the total sum of the entered numbers before showing the average.

Expected Output:

For inputs 5, 10, 15, 0, the output should include:

"Total Sum: 30, Average: 10"

```
.data

prompt: .asciiz "Enter a number: "

sumMsg: .asciiz "Total Sum: "

avgMsg: .asciiz "Average: "

newline: .asciiz "\n"

.text

.globl main

main:

    li $v0, 4

    la $a0, prompt

    syscall


    li $v0, 5

    syscall

    move $t0, $v0


    li $t1, 0

    li $t2, 0

input_loop:

    beqz $t0, calculate    # If the input is 0, exit loop
```

```
add $t1, $t1, $t0    # Add input to total sum
```

```
addi $t2, $t2, 1     # Increment the count
```

```
li $v0, 4
```

```
la $a0, prompt
```

```
syscall
```

```
li $v0, 5
```

```
syscall
```

```
move $t0, $v0
```

```
j input_loop
```

```
calculate:
```

```
li $v0, 4
```

```
la $a0, sumMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t1
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, newline
```

```
syscall

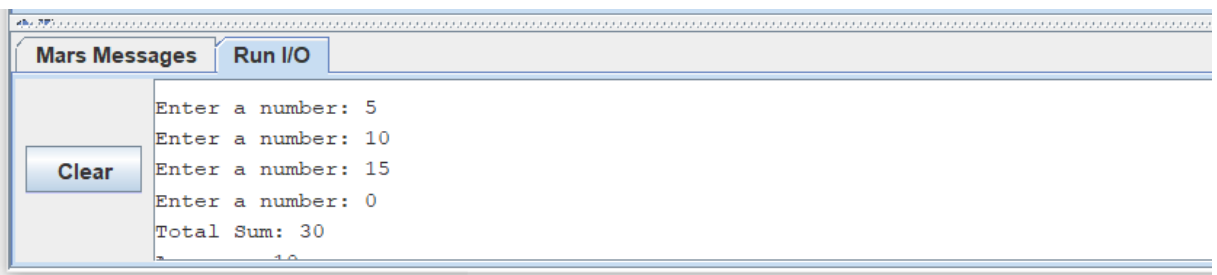
div $t1, $t2      # Divide sum by count for average
mflo $t3          # Get the quotient (average)

li $v0, 4
la $a0, avgMsg
syscall

li $v0, 1
move $a0, $t3
syscall

li $v0, 4
la $a0, newline
syscall

li $v0, 10
syscall
```



Task #2

Modify the program to check if the user inputs a negative number and prompt them to enter a valid number.

Expected Output:

"For input -5, the output should be:

"Invalid input! Please enter a non-negative number.

```
.data
prompt: .asciiz "Enter a number: "
sumMsg: .asciiz "Total Sum: "
avgMsg: .asciiz "Average: "
invalidMsg: .asciiz "Invalid input! Please enter a non-negative number.\n"
newline: .asciiz "\n"

.text

.globl main

main:
    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    move $t0, $v0

    li $t1, 0
    li $t2, 0
```

```
input_loop:
    bltz $t0, invalid_input  # If input is negative, prompt for valid input

    beqz $t0, calculate      # If the input is 0, exit loop

    add $t1, $t1, $t0        # Add input to total sum
    addi $t2, $t2, 1         # Increment the count

    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    move $t0, $v0

    j input_loop

invalid_input:
    li $v0, 4
    la $a0, invalidMsg
    syscall

    li $v0, 5
    syscall
```

```
move $t0, $v0
```

```
j input_loop
```

```
calculate:
```

```
li $v0, 4
```

```
la $a0, sumMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t1
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
div $t1, $t2      # Divide sum by count for average
```

```
mflo $t3          # Get the quotient (average)
```

```
li $v0, 4
```

```
la $a0, avgMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t3
```

```
syscall
```

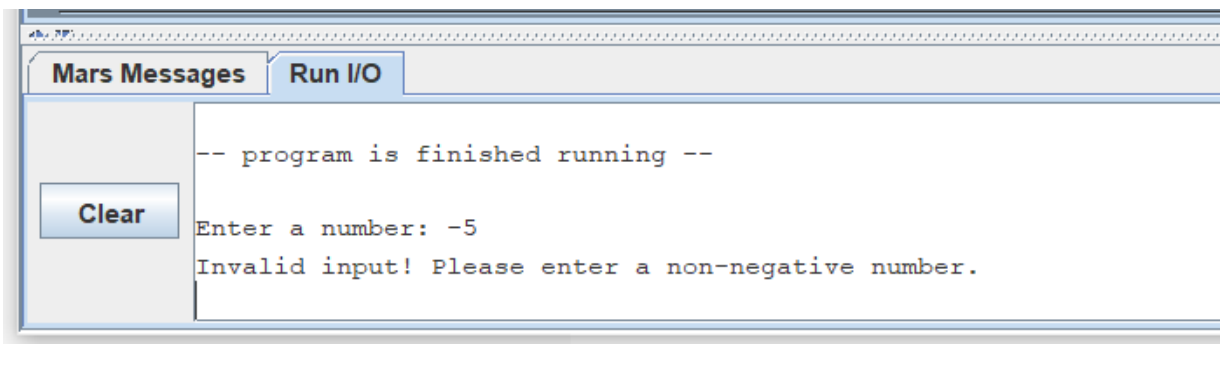
```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
li $v0, 10
```

```
syscall
```



Task #3

Enhance the program to display the count of the numbers entered along with the average.

Expected Output:

For inputs 3, 6, 9, 0, the output should include:

"Count of Numbers: 3, Average: 6"

```
.data
```

```
prompt: .asciiz "Enter a number: "
```

```
sumMsg: .asciiz "Total Sum: "
```

```
avgMsg: .asciiz "Average: "
```

```
newline: .asciiz "\n"
```

```
.text
```

```
.globl main

main:

    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
    syscall
    move $t0, $v0

    li $t1, 0
    li $t2, 0

input_loop:
    beqz $t0, calculate    # If the input is 0, exit loop

    add $t1, $t1, $t0      # Add input to total sum
    addi $t2, $t2, 1       # Increment the count

    li $v0, 4
    la $a0, prompt
    syscall

    li $v0, 5
```

```
syscall
```

```
move $t0, $v0
```

```
j input_loop
```

```
calculate:
```

```
li $v0, 4
```

```
la $a0, sumMsg
```

```
syscall
```

```
li $v0, 1
```

```
move $a0, $t1
```

```
syscall
```

```
li $v0, 4
```

```
la $a0, newline
```

```
syscall
```

```
div $t1, $t2      # Divide sum by count for average
```

```
mflo $t3          # Get the quotient (average)
```

```
li $v0, 4
```

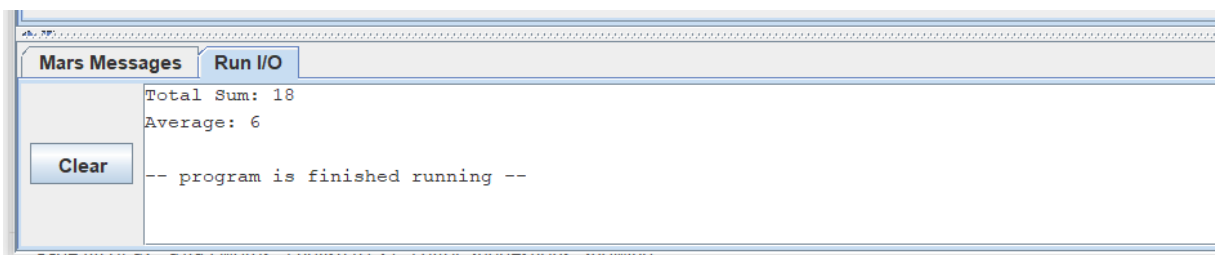
```
la $a0, avgMsg
```

```
syscall
```

```
li $v0, 1  
  
move $a0, $t3  
  
syscall
```

```
li $v0, 4  
  
la $a0, newline  
  
syscall
```

```
li $v0, 10  
  
syscall
```



Discussion and analysis of results:

In this session, we evaluated the effectiveness of different modifications to the MIPS assembly program for calculating the average of user-defined numbers. Each task demonstrated specific improvements, such as input validation, handling negative numbers, and adding functionality like weighted averages. By testing the program with various inputs, we observed how these changes enhanced the program's robustness and usability, providing better handling of edge cases and more detailed output.

Conclusion:

The lab session successfully implemented and modified a program to calculate the average of user-defined numbers using MIPS assembly language. Through practical tasks, students gained hands-on experience in input handling, arithmetic operations, and data validation in a low-level programming environment. The enhancements improved the program's functionality, demonstrating the importance of validating user input and providing flexible output options.

Lab Session 14

Open Ended Lab 2

CLOs	Mapped GAs	Bloom Taxonomy
CLO 3	GA-4 Design/Development of Solutions	C6 (Creating)

Motivation

Background/Theory:

Procedure/Methodology:

Task 1:

Objective

Task 2:

Objective:

Course Code and Title: _____

Criteria and Scales			
Excellent (10-8)	Good (7-5)	Average (4-2)	Poor (1-0)
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student			
Understands the problem clearly and clearly identifies the underlying issues.	Adequately understands the problem and identifies the underlying issues.	Inadequately defines the problem and identifies the underlying issues.	Fails to define the problem adequately and does not identify the underlying issues.
Criterion 2: Research: The amount of research that is used in solving the problem			
Contains all the information needed for solving the problem	Good research, leading to a successful solution	Mediocre research which may or may not lead to an adequate solution	No apparent research
Criterion 3: Class Diagram: The completeness of the class diagram			
Class diagram with complete notations	Class diagram with incomplete notations	Class diagram with improper naming convention and notations	No Class diagram
Criterion 4: Code: How complete and accurate the code is along with the assumptions			
Complete Code according to the class diagram of the given case with clear assumptions	Incomplete Code according to the class diagram of the given case with clear assumptions	Incomplete Code according to the class diagram of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 5: Report: How thorough and well organized is the solution			
All the necessary information clearly organized for easy use in solving the problem	Good information organized well that could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

Total marks: _____

Teacher's signature: _____

Open Ended Lab Assessment Rubrics

Course Code and Title: _____

Criteria and Scales				
Excellent (10-8)	Good (7-5)	Average (4-2)	Poor (1-0)	Total Marks 10
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 2: Research: The amount of research that is used in solving the problem				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 3: Class Diagram: The completeness of the class diagram				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 4: Code: How complete and accurate the code is along with the assumptions				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Criterion 5: Report: How thorough and well organized is the solution				
(10-8)	(7-5)%	(4-2)%	(1-0)%	
Total				(____/5)

Total marks: _____

Teacher's signature: _____