| | |
|---|---|
| **Name:** | Sarvesh Surve |
| **Roll No:** | 73 |
| **Class/Sem:** | SE/IV |
| **Experiment No.:** | 8 |
| **Title:** | Mixed language program to add two numbers |
| **Date of Performance:** | 06/03/2024 |
| **Date of Submission:** | 22/03/2024 |
| **Marks:** | |
| **Sign of Faculty:** | |

**Aim:** Mixed language program for adding two numbers.

**Theory:**

C generates an object code that is extremely fast and compact but it is not as fast as the object code generated by a good programmer using assembly language. The time needed to write a program in assembly language is much more than the time taken in higher level languages like C.

However, there are special cases wher a function is coded in assembly language to reduce the execution time.

Eg: The floating point math package must be loaded assembly language as it is used frequently and its execution speed will have great effect on the overall speed of the program that uses it.

There are also situations in which special hardware devices need exact timing and it is must to write a program in assembly language to meet this strict timing requirement. Certain instructions cannot be executed by a C program

Eg: There is no built in bit wise rotate operation in C. To efficiently perform this it is necessary to use assembly language routine.

Inspite of C being very powerful , routines must be written in assembly language to:

1.Increase the speed and efficiency of the routine

2.Perform machine specific function not available in Microsoft C or Turbo C.

3. Use third party routines

Combining C and assembly:

Built-In-Inline assembles is used to include assembly language routines in C program without any need for a specific assembler.

Such assembly language routines are called in-line assembly.

They are compiled right along with C routines rather than being assembled separately and then linked together using linker modules provided by the C compiler.

Turbo C has inline assembles.

In mixed language program, prefix the keyword asm for a function and write Assembly instruction in the curly braces in a C program

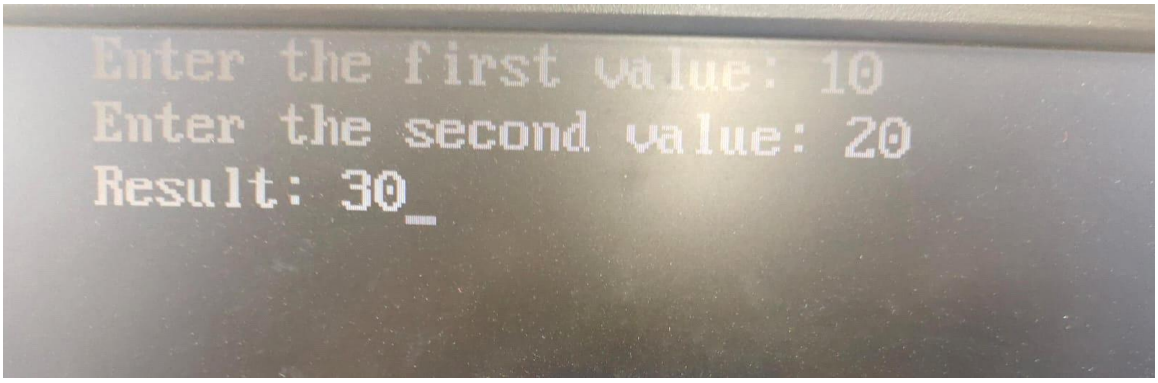
**IMPLEMENTATION :**

CODE :

```
#include <stdio.h>

#include <conio.h>

int main(){

int a,b,c;

clrscr();

printf("Enter a 1st value:");
```

```
scanf("%d",&a);

printf("\nEnter a 2nd value:");

scanf("%d",&b);

asm{

mov ax,a

mov bx,b

SUB ax,bx

mov c,ax

}

printf("result :%d",c);

return 0;

}
```

OUTPUT:




**Conclusion:**
this mixed-language program demonstrates the integration of assembly language with a high-level language like C to achieve low-level operations efficiently, showcasing the versatility and power of mixed-language programming in system-level tasks.

- Explain any 2 branch instructions

Branch instructions are fundamental to computer architecture and assembly language programming. They control the flow of program execution by directing the program to jump to a different location in memory based on certain conditions. Here are explanations of two common branch instructions:

1.Conditional Branch Instruction (e.g., JMP if Zero):

Conditional branch instructions allow the program to change its flow based on the result of a previous operation or the state of certain flags in the processor's status register.

One common example is the "Jump if Zero" (JZ) instruction. This instruction jumps to a specified memory location if the result of the previous operation is zero.

2.Unconditional Branch Instruction (e.g., JMP):

Unconditional branch instructions cause the program to jump to a specified memory location without any condition. They unconditionally alter the flow of program execution.

One common example is the "Jump" (JMP) instruction, which simply transfers control to the specified memory location.

- Explain the syntax of loop.

In assembly language programming, loops are fundamental constructs used to repeat a block of code iteratively until a certain condition is met. The syntax of a loop may vary slightly depending on the specific assembly language, but the general structure remains consistent across different architectures. Here's a breakdown of the syntax of a typical loop:

1.Initialization:

Before entering the loop, initialize loop control variables such as loop counters. This step is typically performed outside the loop.

2.Loop Entry:

The loop entry point marks the beginning of the loop. It's where program execution enters the loop for the first time.

3.Loop Body:

The loop body contains the instructions that are executed repeatedly until the loop termination condition is met. It typically includes the operations you want to repeat.

4.Loop Condition:

The loop condition is evaluated at the beginning of each iteration. If the condition evaluates to true, the loop body is executed. If it evaluates to false, the loop terminates.

5.Loop Termination:

Eventually, the loop termination condition will become true, causing the loop to terminate. Program execution then continues after the loop.