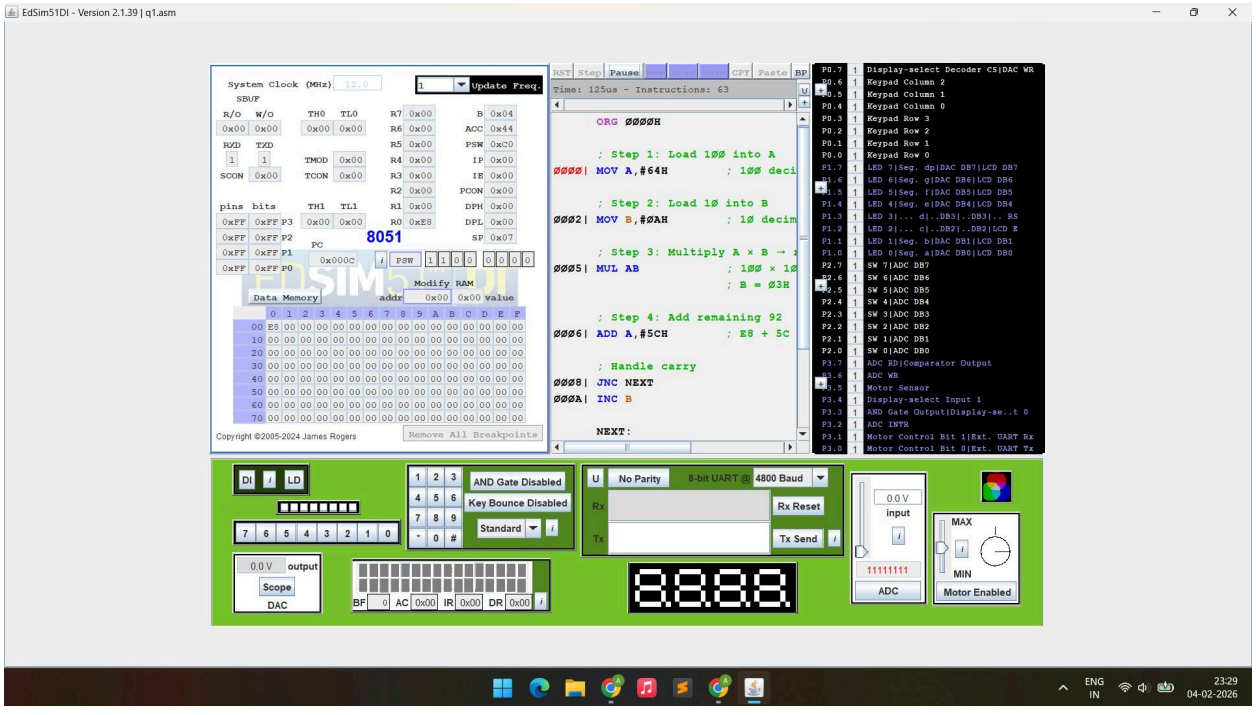
NAME: Abhinav Gunjal
SEM: IV
SEC: B
SUBJECT: Microcontrollers and Embedded Systems

CA-1 ASSIGNMENT

Case study 1: Arithmetic Instructions

Write an 8051 Assembly Language Program (ALP) to generate the last four digits of your PRN using any arithmetic instructions. The program should not directly load the complete PRN number as an immediate value. Instead, it must use appropriate arithmetic operations such as ADD, MUL, or INC to form the number logically. The final result must be stored in the Accumulator register (AX). For example, if a student's PRN is 24070521211, the last four digits are 1211, and the value 1211 should be available in AX at the end of program execution.

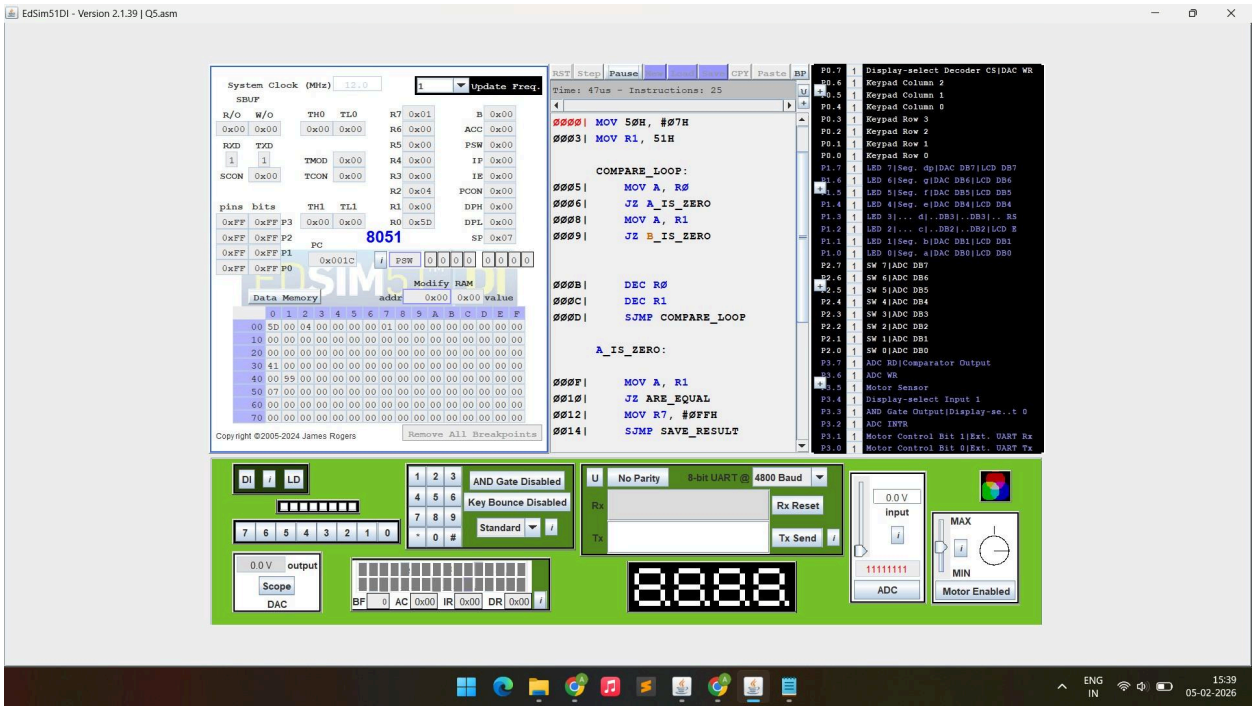


Description:

In this program, the last four digits of my PRN are **1092**, and the number must be generated using arithmetic instructions without directly loading the complete value. Therefore, the number is formed logically using the place value method. First, digit 1 is multiplied by 1000 to obtain 1000, then 0 is multiplied by 100 which gives 0, next 9 is multiplied by 10 to get 90, and finally 2 is added. All these partial results are added step-by-step using arithmetic instructions like MOV, MUL, and ADD. Thus, the number 1092 is constructed logically and the final result is stored in the Accumulator register.

Case Study 2: Instruction Set Limitation Challenge

Execute an 8051-assembly language program for a safety-certified system in which the instructions CJNE, DJNZ, and SUBB are not permitted. Two unsigned numbers are stored in internal RAM locations 50H and 51H. The program must compare these two numbers using only the allowed instruction set (MOV, INC, DEC, JZ, JNZ, CLR, SETB, ANL, ORL) and store the comparison result in a register or memory location such that 01H indicates the value at 50H is greater than the value at 51H, 00H indicates both values are equal, and FFH indicates the value at 50H is less than the value at 51H. The program should be simulated for all three possible cases (A > B, A = B, A < B), and the solution must clearly explain how flag behavior (especially the Zero flag) is utilized to achieve comparison under the given instruction constraints.

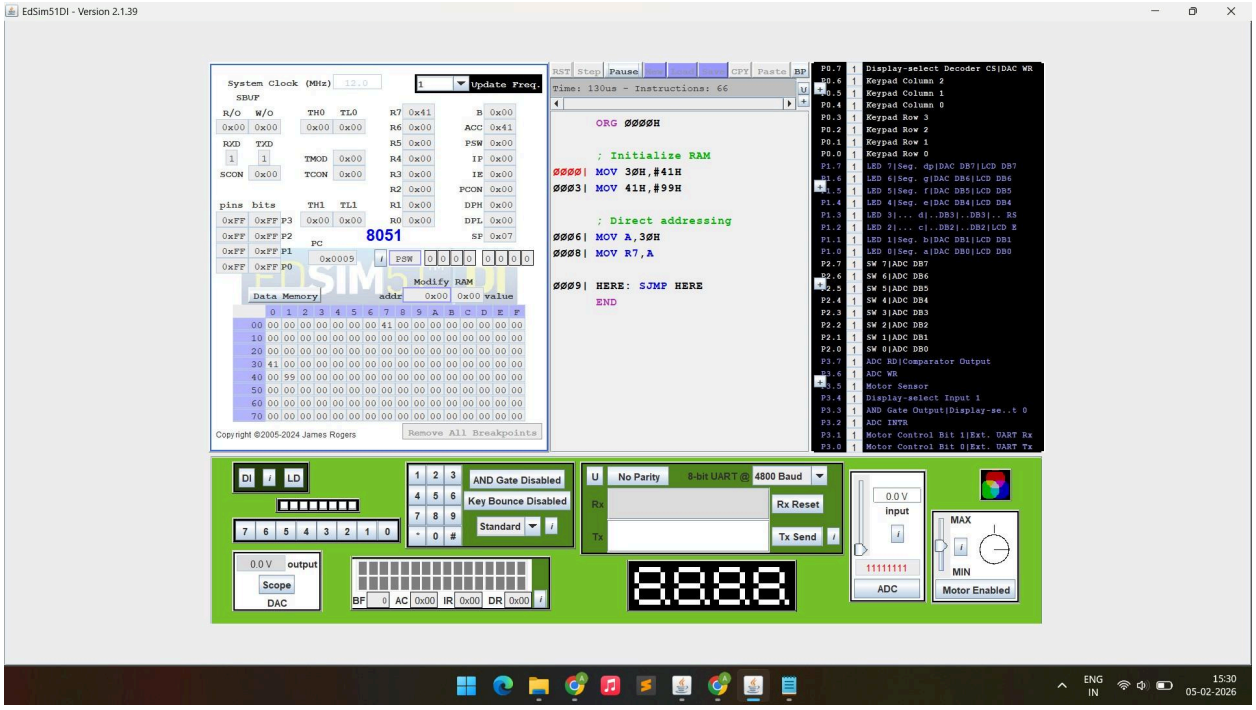


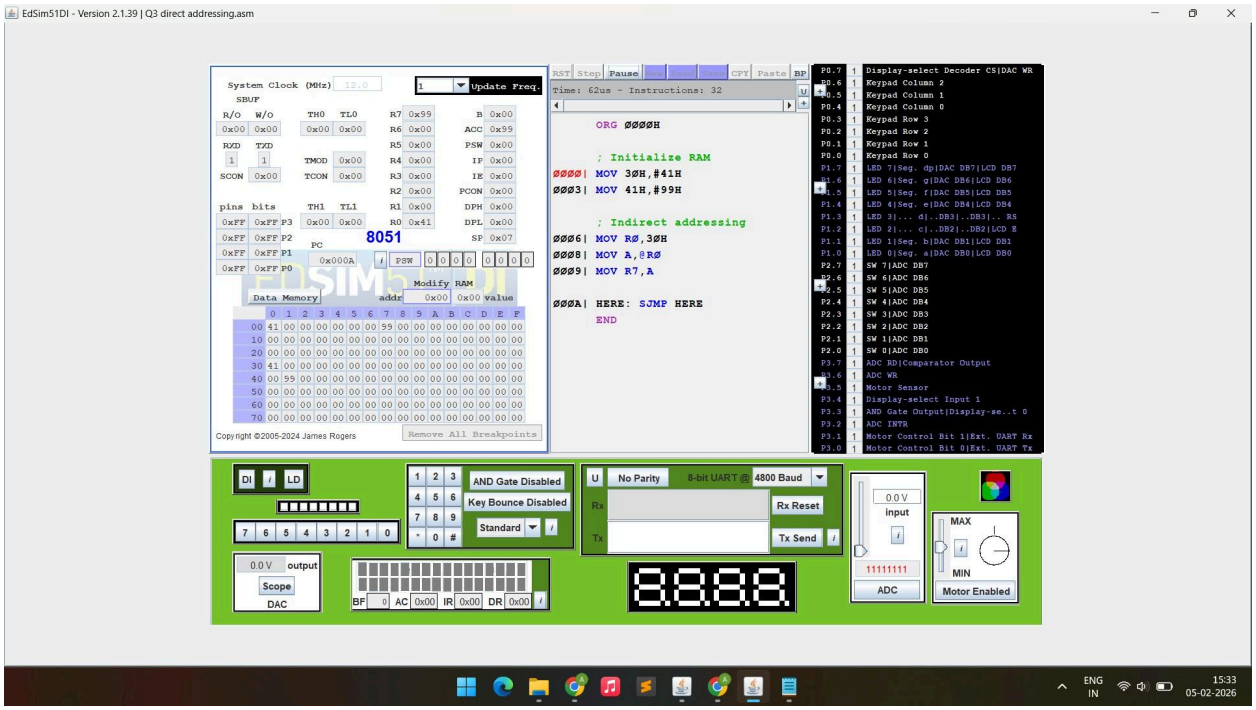
Description:

In this program, two unsigned numbers stored at memory locations 50H and 51H are compared without using CJNE, DJNZ, and SUBB instructions. The comparison is done by repeatedly decrementing both numbers using the DEC instruction and checking the Zero flag after each step. If both numbers become zero at the same time, they are equal and 00H is stored. If the first number remains non-zero while the second becomes zero, then the first number is greater and 01H is stored. If the first becomes zero earlier, then it is smaller and FFH is stored. In this way, the Zero flag is used to perform comparison logically.

Case Study 3: Addressing Mode Puzzle

A student claims that two assembly programs are equivalent because both access the same RAM address; however, this claim is incorrect due to the difference in addressing modes. In this case study, write two short assembly programs—one using direct addressing and the other using indirect addressing—such that both reference the same RAM location. Using an appropriate initial RAM configuration, demonstrate a situation where the outputs of the two programs differ even though the base address is the same. Support the observation with register and RAM snapshots from simulation, and explain that the difference arises because direct addressing accesses the data stored at the given address, whereas indirect addressing treats the contents of that address as a pointer to another memory location, leading to different data being fetched and hence different outputs.



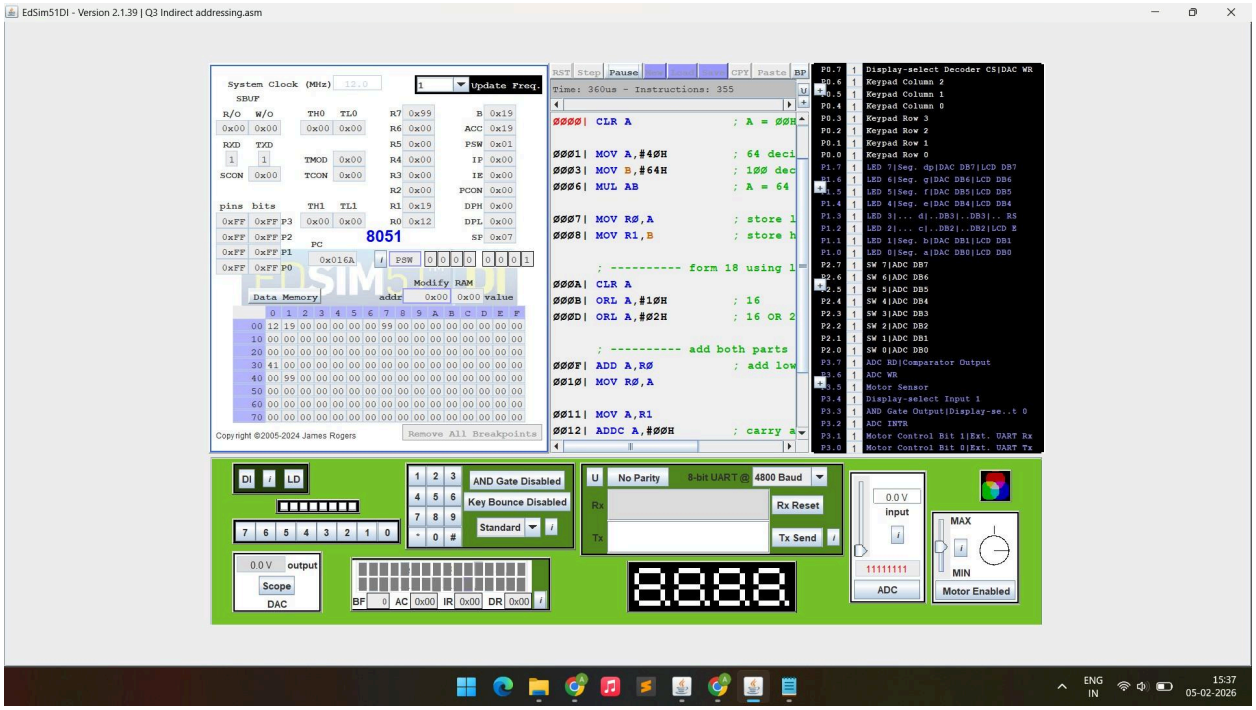


Description:

This program explains the difference between direct and indirect addressing modes even when the same memory address is used. In direct addressing, the instruction directly accesses the data stored at that address. In indirect addressing, the address contains a pointer that refers to another memory location, and the data is fetched from that new location. Because of this pointer behavior, the value obtained may be different. Therefore, even though both programs refer to the same base address, the outputs are different due to different addressing methods.

Case Study 4: Logical Instruction Challenge

Design an 8051 Assembly Language Program in which you must use logical instructions to construct a numeric result. Using multiple logical instructions such as ANL, ORL, and CLR, generate the last four digits of your own mobile number through a suitable sequence of operations (you may split the digits and combine them logically as required). Do not directly load the complete 4-digit number as an immediate value. The program should use more than one logical instruction, and at the end of execution the Accumulator (A) must contain the last four digits of your mobile number. Simulate the program and verify that the final value in the Accumulator matches your mobile number's last four digits.

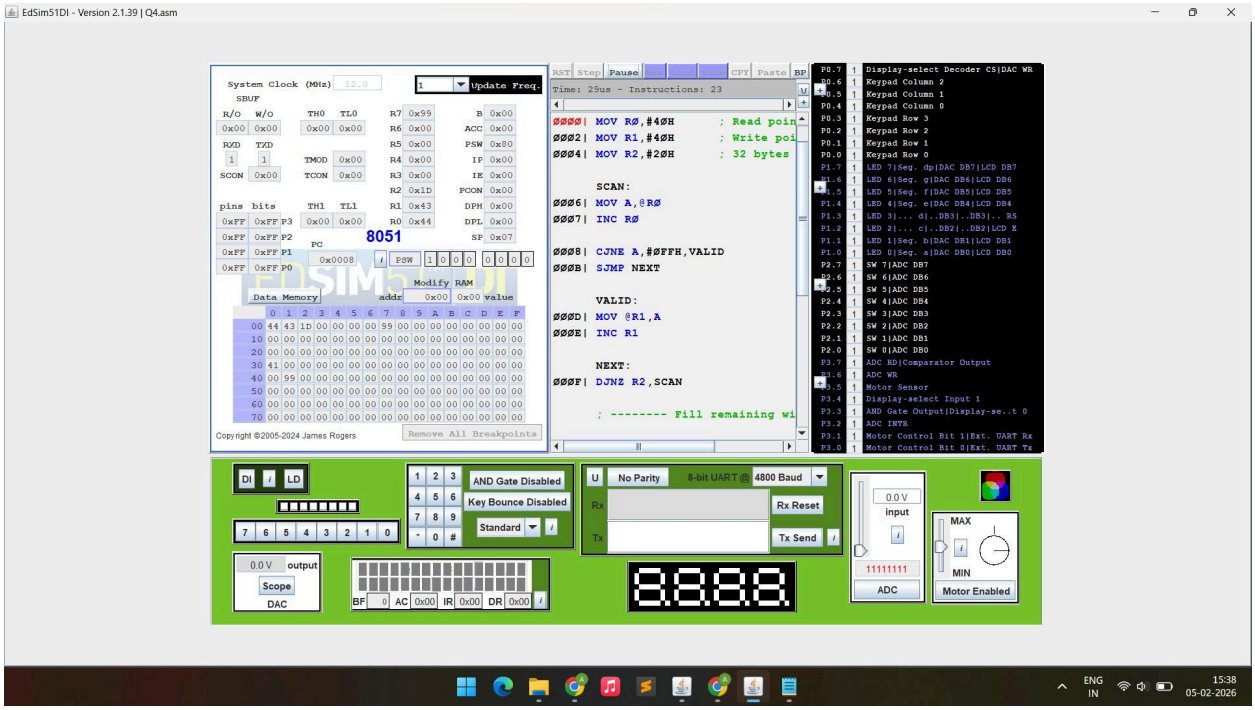


Description

In this program, the last four digits of my mobile number are **6418**, and the number is generated using logical instructions instead of arithmetic operations or direct loading. The accumulator is first cleared using the CLR instruction. Then different parts of the number are created using logical operations like ORL and ANL to combine bit patterns. For example, 6400 and 18 are formed separately and then combined to obtain 6418. This method uses only logical instructions to build the number step-by-step. After execution, the final value 6418 is available in the Accumulator or register pair.

Case Study 5: Memory Mapping under Restriction

An embedded logger stores event codes in internal RAM from 40H to 5FH, but due to strict memory limitations the data must be compacted in-place without using any additional RAM or the stack. Write an assembly language program that scans the memory range 40H–5FH using only indirect addressing, removes all occurrences of the value FFH, shifts the remaining valid data bytes to the left to eliminate gaps, and fills the unused memory locations at the end of the range with 00H. Evaluate the program to show the RAM contents before and after execution, and clearly explain the pointer movement logic used to identify valid data, shift it correctly, and overwrite invalid entries under the given constraints.



Description:

In this program, event codes are stored in memory locations from 40H to 5FH, but some locations contain FFH which are invalid and must be removed. The program scans the memory using indirect addressing and uses two pointers: one for reading data and one for writing valid data. Whenever a value other than FFH is found, it is copied to the next valid position. Invalid values are skipped. After shifting all valid data to the left, the remaining empty locations are filled with 00H. This process removes unwanted data and compacts memory efficiently without using extra memory or stack.