Embedded Systems II - Software-Based Laboratory Exercises

This repository contains the assembly language source code for a series of laboratory exercises for an Embedded Systems II course based on the 8051 microcontroller. The labs are designed to be completed using a software simulator, with no hardware required.

Overview of Labs

Lab 1: 8051 Assembly Fundamentals

This lab introduces the fundamentals of 8051 assembly language programming. It covers basic concepts such as:

- Data movement and register manipulation.
- Looping and iteration with 16-bit counters.
- Basic I/O operations to control LEDs.
- The use of subroutines for code modularity.

Lab 2: Basic I/O and Port Addressing

This lab delves deeper into I/O operations, with a focus on:

- Bit-addressable I/O for fine-grained control of individual port pins.
- Reading digital inputs from switches.
- Implementing a software-based debouncing routine to handle the mechanical bouncing of switches.

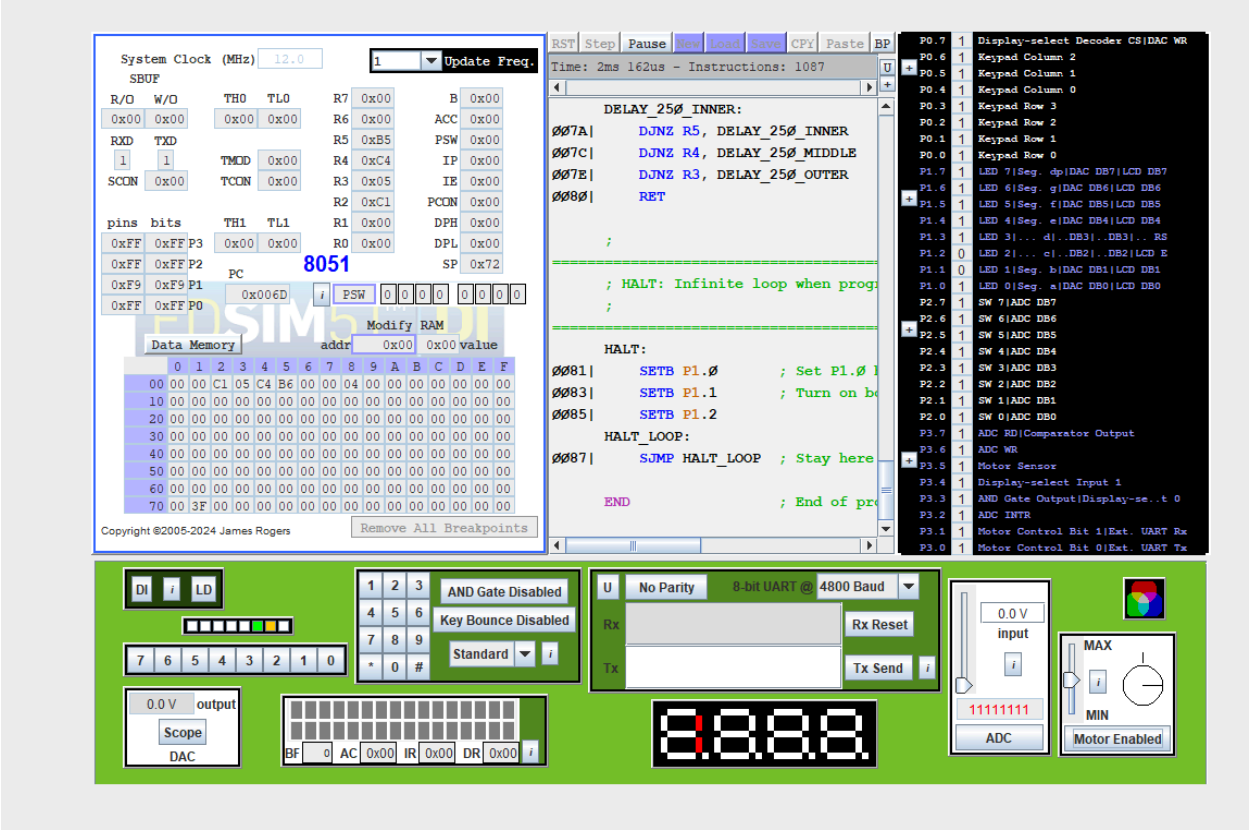
Lab 3: Analog-Digital Interaction

This lab covers the simulation of analog-to-digital conversion and interfacing with a character LCD:

- Simulating an external ADC to read analog values.
- Using timer and external interrupts to manage the ADC conversion process.
- Interfacing with a Hitachi HD44780-compatible character LCD in 4-bit mode.
- Creating custom characters (CGRAM) to display a bar graph representation of the analog value.

Simulator

The labs are designed to be run on an 8051 simulator. This repository includes the `edsim51di.jar` simulator.



How to Run the Simulator

1. **Java Requirement:** Ensure you have a Java Runtime Environment (JRE) installed on your system.

2. **Navigate to the Simulator Directory:** Open a terminal or command prompt and navigate to the `Simulator_For_The_Labs` directory.

3. **Run the Simulator:** Execute the following command:

   ```
   java -jar edsim51di.jar
   ```

4. **Load a Program:** Once the simulator is open, you can load one of the `.asm` files from the lab directories to assemble and run it.

Alternatively, you can use other 8051 simulators, such as the online simulator at https://8051-simulator.vercel.app/, as mentioned in the lab manual. You may need to copy and paste the code into the online simulator's editor.

Lab 1: 8051 Assembly Fundamentals in a Simulator

This lab explores the fundamentals of 8051 assembly language programming using a simulator. The exercises cover data movement, looping, I/O operations, and subroutines.

Files

- `LAB1_1000_ITERATION.asm`: An assembly program that increments a register (R0) in a loop for 1000 iterations.
- `LAB1_LED_BLINK_SUB.asm`: An assembly program that demonstrates the use of subroutines to blink two LEDs connected to P1.0 and P1.1.
- `LAB1_TOGGLE_P1_0.asm`: An assembly program that toggles the state of a single LED connected to P1.0.

Concepts Implemented

1. Looping and Iteration

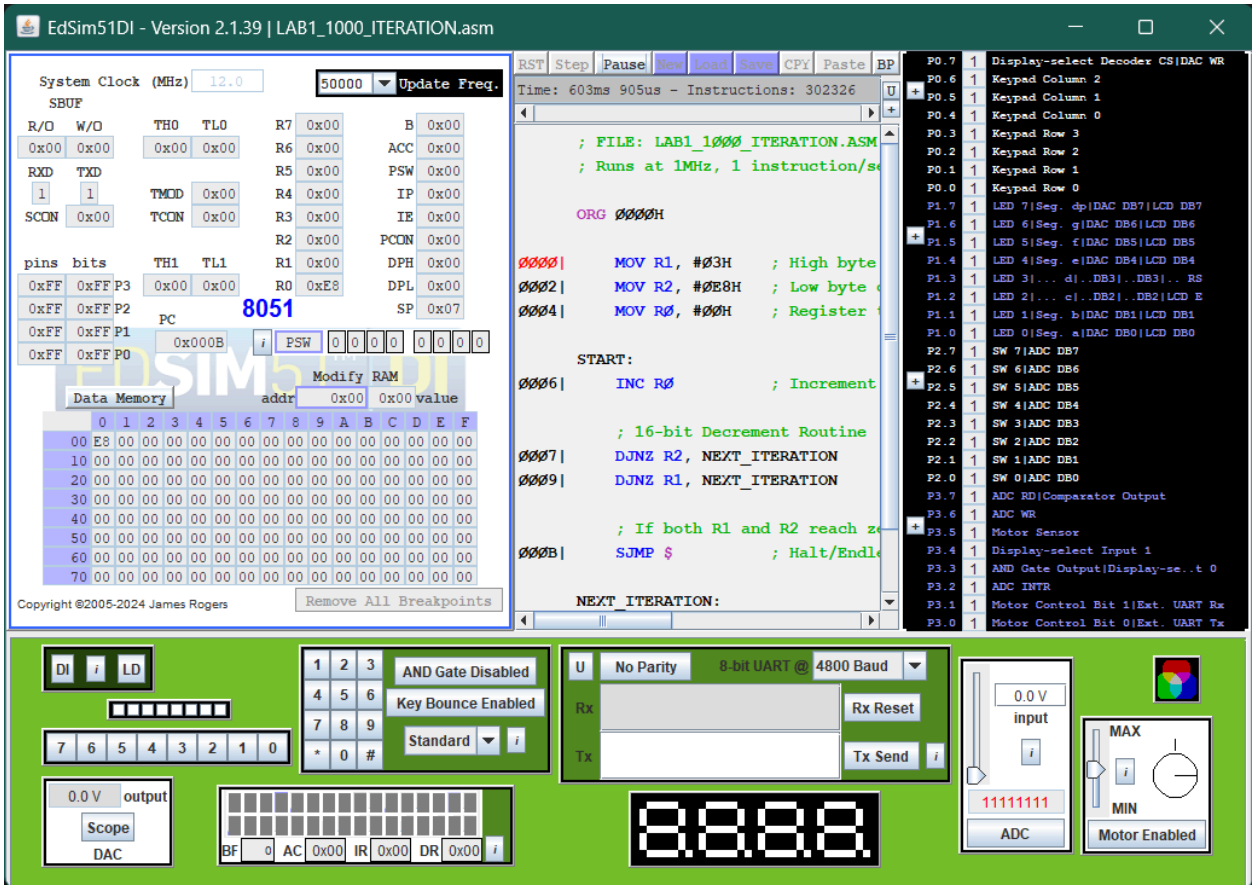
The `LAB1_1000_ITERATION.asm` program demonstrates a 16-bit counter to achieve 1000 iterations.

Code Snippet (`LAB1_1000_ITERATION.asm`):

```
; FILE: LAB1_1000_ITERATION.ASM
; Runs at 1MHz, 1 instruction/sec

ORG 0000H

    MOV R1, #03H    ; High byte of 1000 (03E8H)
    MOV R2, #0E8H   ; Low byte of 1000 (E8H)
    MOV R0, #00H    ; Register to be incremented

START:
    INC R0          ; Increment R0 (The primary action)

    ; 16-bit Decrement Routine (R1:R2)
    DJNZ R2, NEXT_ITERATION
    DJNZ R1, NEXT_ITERATION

    ; If both R1 and R2 reach zero, the loop halts.
    SJMP $          ; Halt/Endless loop (Equivalent to MOV PC, PC on some
systems)

NEXT_ITERATION:
    SJMP START
END
```

- **Explanation:** This program initializes `R1` and `R2` with the high and low bytes of 1000 (0x03E8). The `DJNZ` (Decrement and Jump if Not Zero) instruction is used to create a

loop. The inner loop decrements `R2` and the outer loop decrements `R1`. `R0` is incremented in each iteration. The program halts in an endless loop (`SJMP $`) after 1000 iterations.



2. Subroutines and I/O Operations

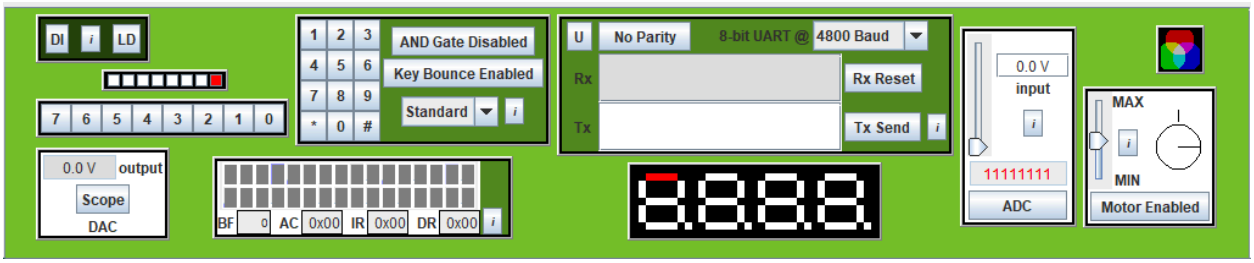
The `LAB1_LED_BLINK_SUB.asm` program demonstrates how to use subroutines (`CALL` and `RET`) to structure code and control I/O pins to blink LEDs.

Code Snippet (`LAB1_LED_BLINK_SUB.asm`):

```
; FILE: LAB1_LED_BLINK_SUB.ASM
; Runs at 1MHz, 1 instruction/sec
ORG 0000H

START:
    CALL BLINK_SEQUENCE
    SJMP START

; Subroutine to sequentially blink two LEDs (P1.0 and P1.1)
BLINK_SEQUENCE:
    ; --- Blink LED 0 (P1.0) ---
    CLR P1.0        ; Turn LED 0 ON (Logic 0)
    SETB P1.1       ; Turn LED 1 OFF (Logic 1)
```

```
    ; --- Blink LED 1 (P1.1) ---
    CLR P1.1        ; Turn LED 1 ON (Logic 0)
    SETB P1.0       ; Turn LED 0 OFF (Logic 1)

    RET             ; Return from Subroutine
END
```

- **Explanation:** The `BLINK_SEQUENCE` subroutine is called from the `START` label. This subroutine first turns on the LED at P1.0 and turns off the LED at P1.1. Then it does the reverse. The `RET` instruction returns the program control to the instruction after `CALL`. The main program then jumps back to `START`, creating an infinite loop.



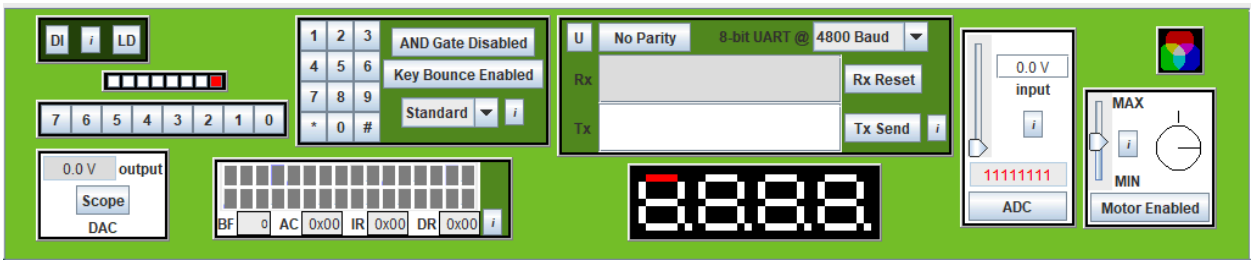
3. Basic I/O Toggling

The `LAB1_TOGGLE_P1_0.asm` program shows the simplest way to toggle an I/O pin.

Code Snippet (`LAB1_TOGGLE_P1_0.asm`):

```
; FILE: LAB1_TOGGLE_P1_0.ASM
; Runs at 1MHz, 1 instruction/sec
ORG 0000H

START:
    CPL P1.0        ; Toggle LED 0 (P1.0)
    SJMP START
END
```

5. **Explanation:** The `CPL P1.0` instruction complements the value of the pin P1.0. If it's 1, it becomes 0, and vice-versa. The `SJMP START` creates an infinite loop to continuously toggle the LED. Assuming a 1MHz clock and 1 instruction per cycle, this would toggle the pin at a very high frequency. The comment in the file suggests a 1 instruction/sec execution which would result in a 0.5Hz toggle rate.



Simulator and Waveforms

To run these programs, you can use the provided `edsim51di.jar` simulator or any other 8051 simulator. The lab specifications suggest observing the waveforms. For example, in `LAB1_TOGGLE_P1_0.asm`, the waveform for P1.0 would be a square wave.

Reflection on Instruction Timings and Cycles

The timing of these programs is highly dependent on the clock frequency of the simulated 8051 microcontroller. For instance, in `LAB1_TOGGLE_P1_0.asm`, the toggle frequency of P1.0 is determined by the execution time of the `CPL` and `SJMP` instructions. If the simulator runs at 1MHz and each instruction takes one machine cycle, the loop takes two cycles. This results in a high-frequency square wave on P1.0.

Lab 2: Basic I/O and Port Addressing

This lab focuses on I/O operations, specifically bit-addressing and handling input with debouncing. The exercises demonstrate how to manipulate individual bits of a port and how to read input from switches reliably.

Files

- `LAB2_BIT_ADDRESSING.asm`: This program demonstrates various ways to manipulate individual bits of I/O ports.
- `LAB2_ECHO_W_DEBOUNCE.asm`: This program reads the state of switches on Port 2 and echoes it to LEDs on Port 1, with a debounce delay to handle mechanical switch bouncing.

Concepts Implemented

1. Bit-Level I/O Operations

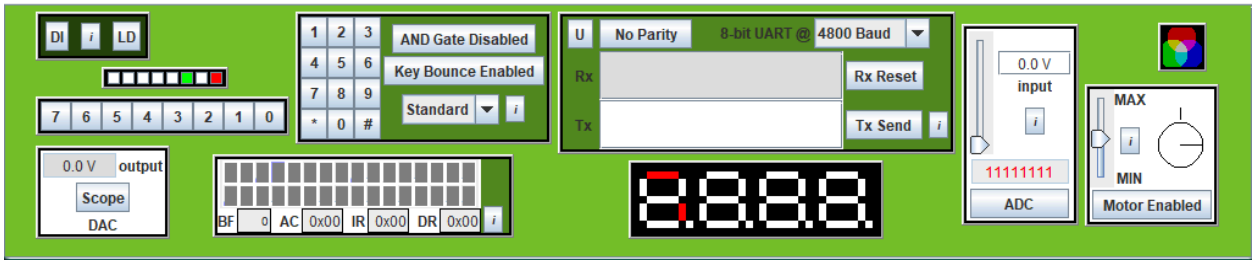
The 8051 provides instructions to manipulate individual bits of certain Special Function Registers (SFRs) and I/O ports. `LAB2_BIT_ADDRESSING.asm` shows how to use these instructions.

Code Snippet (`LAB2_BIT_ADDRESSING.asm`):

```
; FILE: LAB2_BIT_ADDRESSING.ASM
; Runs at 1MHz, 1 instruction/sec

ORG 0000H

START:
    ; 1. Set/Clear a bit directly (LED 0 ON/OFF)
    SETB P1.0       ; P1.0 = 1 (LED 0 OFF)
    CLR P1.7        ; P1.7 = 0 (LED 7 ON)
```

```
    ; 2. Move data to a bit using the Carry Flag (CY)
    MOV C, P1.0     ; Move the state of P1.0 to the Carry Flag (C)
    MOV P1.1, C     ; Copy the value of C (1) to P1.1 (LED 1 OFF)

    ; 3. Use the SETB/CLR operations to invert the state
    CPL C           ; Complement the Carry flag (C = 0)
    MOV P1.2, C     ; Copy C (0) to P1.2 (LED 2 ON)

    CPL P1.0       ; Reset state before looping
    CPL P1.7
    SJMP START
END
```

- **Explanation:**
  - `SETB P1.0`: Sets the bit P1.0 (Port 1, bit 0) to 1.
  - `CLR P1.7`: Clears the bit P1.7 (Port 1, bit 7) to 0.
  - `MOV C, P1.0`: Copies the value of P1.0 to the Carry Flag (C).
  - `MOV P1.1, C`: Copies the value of the Carry Flag to P1.1.
  - `CPL C`: Complements the Carry Flag. This demonstrates the fine-grained control over individual I/O pins.



2. Input Debouncing

Mechanical switches often "bounce" when pressed or released, causing multiple rapid transitions. The `LAB2_ECHO_W_DEBOUNCE.asm` program implements a delay-based debouncing routine.

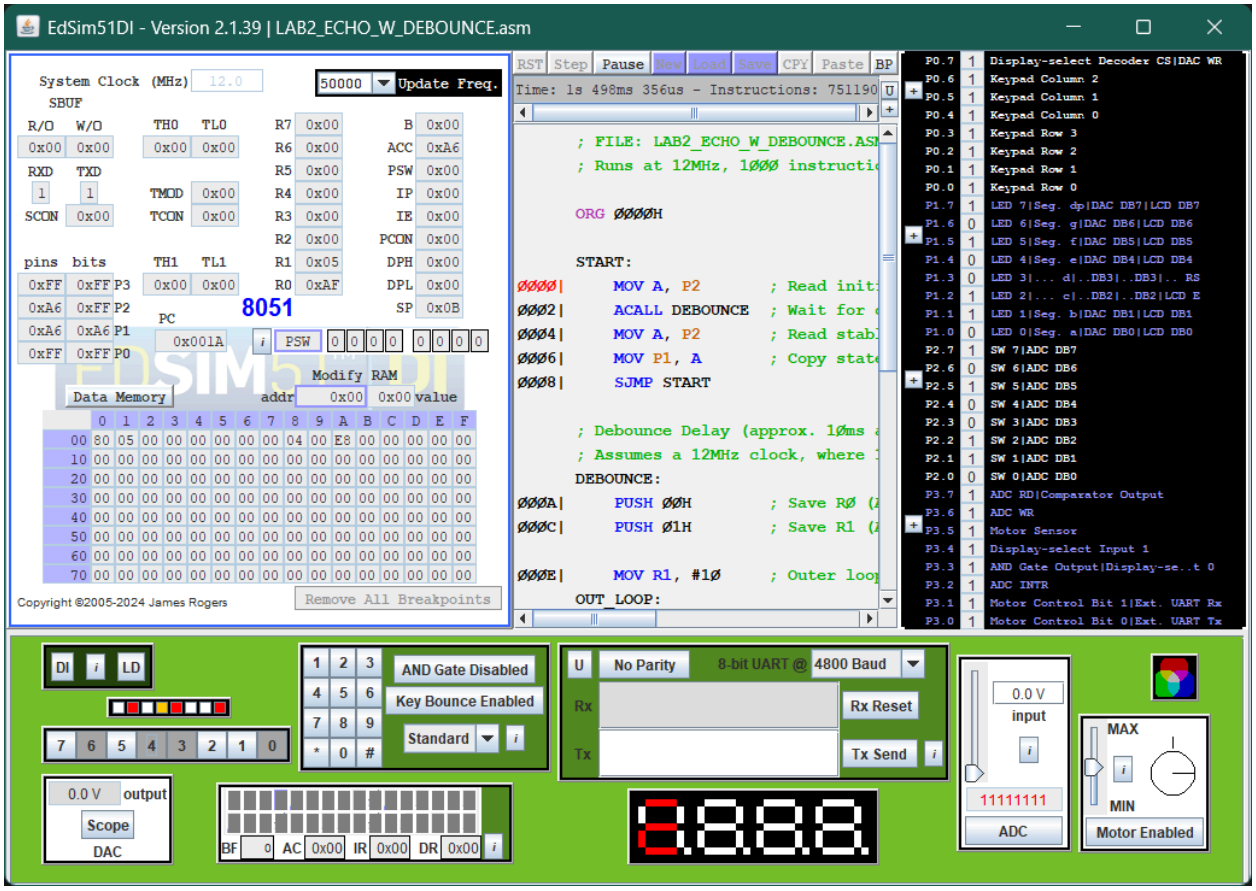
Code Snippet (`LAB2_ECHO_W_DEBOUNCE.asm`):

```
; FILE: LAB2_ECHO_W_DEBOUNCE.ASM
; Runs at 12MHz, 1000 instruction/sec or higher

ORG 0000H

START:
    MOV A, P2       ; Read initial state of switches from Port 2
    ACALL DEBOUNCE  ; Wait for contacts to settle
    MOV A, P2       ; Read stable state of switches
    MOV P1, A       ; Copy state to LEDs on Port 1
    SJMP START
```

```
; Debounce Delay (approx. 10ms at 12MHz)
; Assumes a 12MHz clock, where 1 machine cycle is 1us.
DEBOUNCE:
    PUSH 00H        ; Save R0 (Address 00H)
    PUSH 01H        ; Save R1 (Address 01H)

    MOV R1, #10     ; Outer loop (10 repeats)
OUT_LOOP:
    MOV R0, #200    ; Inner loop (200 repeats)
IN_LOOP:
    DJNZ R0, IN_LOOP
    DJNZ R1, OUT_LOOP

    ; Total Delay: 10 * (200 * 2us/cycle) = 4ms (approx.)
    ; Let's adjust for ~10ms for better visibility in simulator or for a
real-world scenario
    MOV R1, #50     ; Let's use 50 repetitions for ~10ms delay (50 * 200 *
2us = 20ms)
OUT_LOOP_2:
    MOV R0, #200
IN_LOOP_2:
    DJNZ R0, IN_LOOP_2
    DJNZ R1, OUT_LOOP_2

    POP 01H         ; Restore R1 (Address 01H)
    POP 00H         ; Restore R0 (Address 00H)
    RET
END
```

6. **Explanation:**
    1. The program reads the state of Port 2 into the accumulator.
    2. It calls the `DEBOUNCE` subroutine.
    3. The `DEBOUNCE` subroutine is a delay loop. It uses nested loops with `DJNZ` to wait for a certain amount of time (~10-20ms). This allows the switch contacts to settle into a stable state.
    4. After the delay, the program reads the state of Port 2 again, which should now be stable.
    5. The stable state is then written to Port 1, lighting up the corresponding LEDs.

    – The `PUSH` and `POP` instructions are used to save and restore the values of registers `R0` and `R1` that are used in the delay routine. This is good practice to avoid side effects in the main program.



Simulator and LED Patterns

When running `LAB2_ECHO_W_DEBOUNCE.asm` in a simulator, you can connect virtual switches to Port 2 and LEDs to Port 1. When you change the state of the switches, you will see the corresponding LEDs on Port 1 light up after a short delay.

Lab 3: Analog-Digital Interaction (Software Simulation)

This lab explores the interface between an 8051 microcontroller and an external Analog-to-Digital Converter (ADC) and a character LCD. The program reads an analog value from the ADC and displays it as a bar graph on the LCD.

Files

- `LAB3_ADC_TO_LED.asm`: An assembly program that reads a value from a simulated ADC, converts it to a bar graph representation, and displays it on a character LCD.

Concepts Implemented

1. Interfacing with a Character LCD

The program initializes and communicates with a Hitachi HD44780-compatible character LCD in 4-bit mode. This involves sending a series of commands to configure the display and then sending character data to be displayed.

Key LCD Operations: * **Initialization:** The `main` routine contains a sequence of instructions to initialize the LCD, setting it to 4-bit mode, clearing the display, and setting the entry mode. * **Sending Commands and Data:** The `sendCharacter` subroutine sends a byte of data or a command to the LCD. It splits the byte into two nibbles and sends them sequentially. The `RS` pin (P1.3) is used to differentiate between commands (`RS=0`) and data (`RS=1`). * **Custom Characters (CGRAM):** The program creates custom characters for the bar graph by writing pixel patterns to the Character Generator RAM (CGRAM) of the LCD. The `sendPattern` subroutine is used for this purpose.

2. Analog-to-Digital Conversion (ADC)

The program simulates the operation of an external ADC. * **Starting a Conversion:** The `timer0ISR` is configured to trigger every 200µs. It generates a pulse on the ADC's `WR` (Write) pin (P3.6) to start a new conversion. * **Reading the Conversion Result:** The `ext0ISR` is an external interrupt service routine that is triggered when the ADC conversion is complete (signaled by the ADC's `INTR` pin connected to the 8051's `INT0`). This ISR reads the digital value from the ADC (connected to Port 2) into the `B` register.

3. Interrupts

This program makes extensive use of interrupts for timing and handling external events. * **Timer 0 Interrupt:** Used to periodically start the ADC conversion. This ensures that the analog input is sampled at a regular interval. * **External 0 Interrupt:** Used to detect the end of an ADC conversion. This is an efficient way to handle the ADC, as the microcontroller can perform other tasks while the conversion is in progress.

4. Bar Graph Display Logic

The `updateBarGraph` subroutine converts the 8-bit digital value from the ADC into a 16-level bar graph on the 2-line LCD. * The top 4 bits of the ADC result are used to determine the height of the bar graph. * The most significant bit of the ADC result determines whether the bar is displayed on the top or bottom line of the LCD. * The lower 3 bits determine which of the 8 custom bar graph characters to display.

Code Snippet (`LAB3_ADC_TO_LED.asm` - Interrupt Service Routines):

```
; timer 0 ISR - simply starts an ADC conversion
timer0ISR:
    CLR P3.6 ; clear ADC WR line
    SETB P3.6 ; then set it - this results in the required positive edge to
start a conversion
```

```
    RETI ; return from interrupt

; external 0 ISR - responds to the ADC conversion complete interrupt
ext0ISR:
    CLR P3.7 ; clear the ADC RD line - this enables the data lines
    MOV B, P2 ; move ADC outputs to B
    SETB P3.7 ; disable the ADC data lines by setting RD
    CALL updateBarGraph ; update the bar graph using the new reading from the
ADC
    RETI ; return from interrupt
```

- **Explanation:** This shows the core of the ADC interaction. The timer interrupt starts the conversion, and the external interrupt reads the result and updates the display. This decouples the ADC process from the main program flow.

Simulator and Output

The output on the LCD would be a bar graph that changes in height according to the analog input value provided to the ADC.