EMBEDDED SYSTEMS LAB REPORT

Lab 4: Architecture and Memory Layout (Simulation)

Simulator Used: EDSIM51DI (8051 Simulator)

1. INTRODUCTION

This laboratory exercise explores the fundamental architecture and memory organization of the 8051 microcontroller. Understanding how a microcontroller organizes and accesses memory is critical for embedded systems programming, as it directly affects program design, efficiency, and debugging.

The 8051 microcontroller implements a Harvard architecture, which physically separates code (program) memory from data memory. This contrasts with the Von Neumann architecture, where a single memory space stores both instructions and data. This lab uses the EDSIM51DI simulator to visualize memory maps, demonstrate different memory access methods, and observe stack behavior during normal operations and interrupt service routines (ISRs).

Through hands-on simulation, we will examine how the 8051 distinguishes between code and data memory, how the stack grows during function calls and interrupts, and the practical implications of these architectural choices.

2. OBJECTIVES

The primary objectives of this laboratory are:

1. **Compare Von Neumann and Harvard architectures** and understand their fundamental differences
2. **Explore the 8051 microcontroller memory map**, identifying code memory, data memory, and special function registers
3. **Demonstrate code versus data access** using appropriate 8051 assembly instructions
4. **Observe stack growth** during PUSH/POP operations and function calls
5. **Analyze the impact of interrupt service routines (ISRs)** on stack usage
6. **Create comparative diagrams** outlining memory regions and access characteristics

3. THEORY

3.1 Von Neumann vs Harvard Architecture

Von Neumann Architecture:

- Uses a **single memory space** for both program instructions and data
- Instructions and data share the **same bus**
- Simpler design but can create a bottleneck (the "Von Neumann bottleneck") when the processor needs to fetch instructions and data simultaneously
- Examples: Most general-purpose computers, early microprocessors

Harvard Architecture:

- Uses **separate memory spaces** for program code and data
- **Separate buses** for instruction fetch and data access
- Allows **simultaneous access** to instructions and data, improving performance
- More complex but eliminates the Von Neumann bottleneck
- Examples: 8051, PIC microcontrollers, DSP processors

3.2 Why the 8051 Uses Harvard Architecture

The 8051 microcontroller implements a modified Harvard architecture for several reasons:

1. **Performance:** Separate buses allow the CPU to fetch the next instruction while executing the current one
2. **Security:** Code memory can be protected from accidental modification during runtime
3. **Memory expansion:** Different types of memory (ROM for code, RAM for data) can be optimized independently
4. **Embedded applications:** Most embedded programs are fixed and don't require self-modifying code

4. 8051 MEMORY MAP EXPLANATION

The 8051 has a well-defined memory organization with distinct regions:

4.1 Code Memory (Program Memory)

- **Address Range:** 0000H to FFFFH (up to 64KB)
- **Purpose:** Stores program instructions (the actual code)
- **Access Method:** Uses `MOVC` (Move Code) instruction
- **Type:** Typically ROM, EPROM, or Flash memory
- **Characteristics:** Read-only during normal execution

4.2 Data Memory

The 8051 has two types of data memory:

Internal RAM (128 bytes in standard 8051):

- **Address Range:** 00H to 7FH
- **Subdivisions:**
  - 00H-1FH: Register banks (4 banks of 8 registers each)
  - 20H-2FH: Bit-addressable area (16 bytes = 128 bits)
  - 30H-7FH: General-purpose RAM (scratch pad area)
- **Access Method:** Direct or indirect addressing using `MOV`

External RAM (optional):

- **Address Range:** 0000H to FFFFH (up to 64KB)
- **Access Method:** Indirect addressing using `MOVX` instruction
- **Purpose:** Additional data storage when internal RAM is insufficient

4.3 Special Function Registers (SFRs)

- **Address Range:** 80H to FFH
- **Purpose:** Control registers for peripherals (timers, ports, UART, etc.)
- **Access Method:** Direct addressing using `MOV`

4.4 Stack

- **Location:** Resides in internal RAM
- **Stack Pointer (SP):** Special register that points to the top of the stack
- **Default SP Value:** 07H (stack starts at 08H)
- **Growth Direction:** Grows upward (toward higher addresses)
- **Usage:** Stores return addresses, register values during ISRs, and local variables

5. PRACTICAL TASKS EXPLANATION

We developed four separate assembly programs to demonstrate different aspects of 8051 memory architecture:

5.1 Module 1: Code Memory Access

Purpose: Demonstrate how to read data stored in code memory using the `MOVC` instruction.

Concept: In embedded systems, constant data (like lookup tables, text strings, or calibration values) is often stored in code memory to save precious RAM space.

How it works:

- A lookup table is defined in code memory using the `DB` (Define Byte) directive
- The `MOVC A, @A+DPTR` instruction fetches data from code memory
- The DPTR (Data Pointer) holds the base address of the table
- The accumulator (A) holds the offset into the table

5.2 Module 2: Data Memory Access

Purpose: Demonstrate normal data memory operations using `MOV` instructions.

Concept: Data memory is used for variables that change during program execution.

How it works:

- Variables are stored in internal RAM (addresses 30H-7FH)
- The `MOV` instruction transfers data between registers and memory
- Both direct and indirect addressing can be used

5.3 Module 3: Stack Growth Demonstration

Purpose: Visualize how the stack grows when data is pushed and shrinks when data is popped.

Concept: The stack is essential for function calls, temporary storage, and interrupt handling.

How it works:

- The Stack Pointer (SP) register tracks the top of the stack
- `PUSH` increments SP and stores data at the new location
- `POP` retrieves data and decrements SP
- We observe SP changes and memory contents during these operations

5.4 Module 4: ISR Stack Usage

Purpose: Demonstrate automatic stack usage during interrupt service routines.

Concept: When an interrupt occurs, the processor automatically saves the Program Counter (return address) on the stack before jumping to the ISR.

How it works:

- An interrupt is configured (we use Timer 0)
- When the interrupt fires, the return address is automatically pushed onto the stack
- The ISR executes
- The `RETI` instruction automatically pops the return address and returns to the main program

6. CODE

Below are four independent modules. Each can be loaded and run separately in EDSIM51DI.

MODULE 1: Code Memory Access (MOVC)

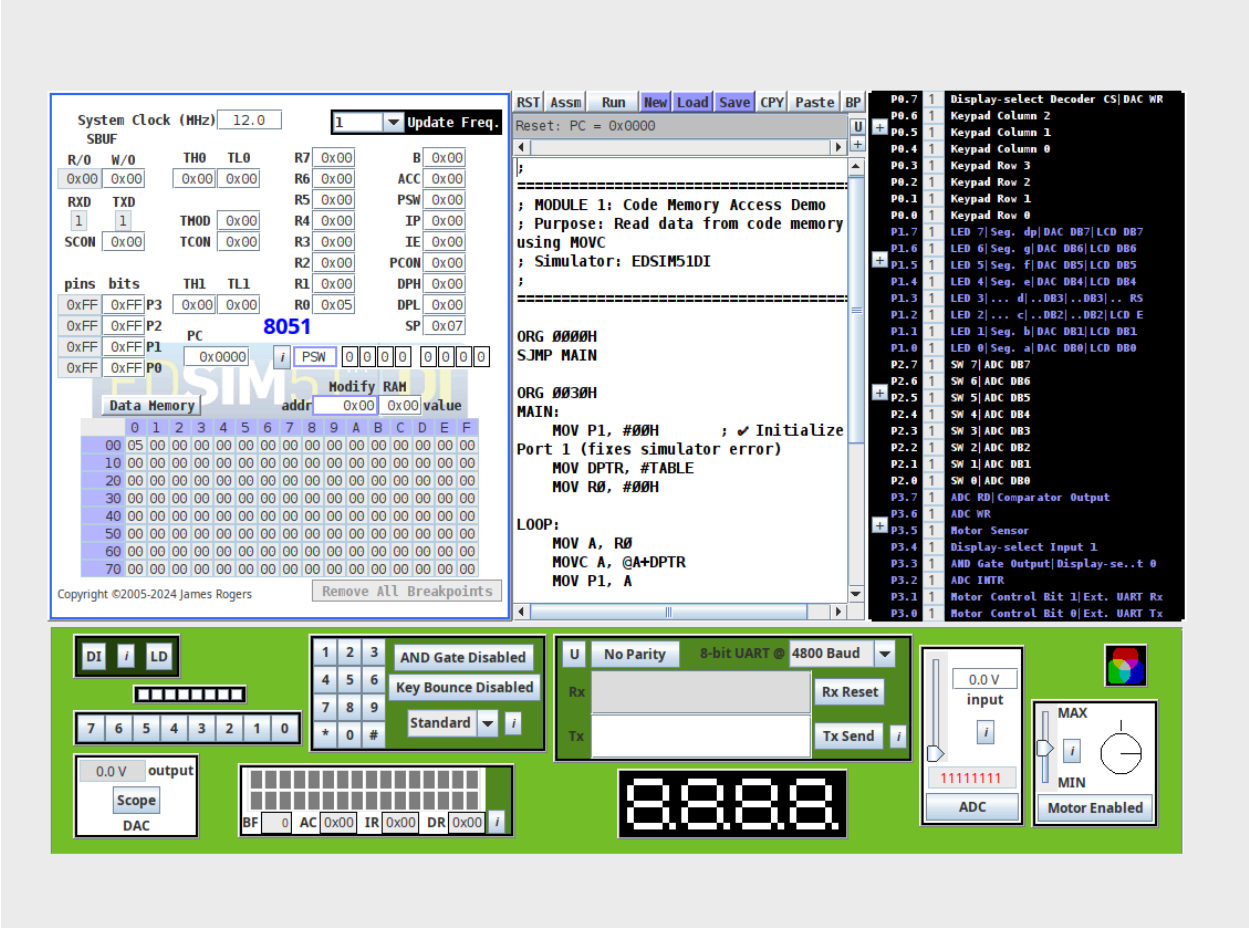
```
    ; ======================================
     ; MODULE 1: Code Memory Access Demo
     ; Purpose: Read data from code memory using MOVC
     ; Simulator: EDSIM51DI
     ; ======================================

     ORG 0000H
0000| SJMP MAIN

     ORG 0030H
     MAIN:
0030|     MOV P1, #00H
0033|     MOV DPTR, #TABLE
0036|     MOV R0, #00H

     LOOP:
0038|     MOV A, R0
0039|     MOVC A, @A+DPTR
003A|     MOV P1, A

003C|     INC R0
003D|     CJNE R0, #05H, LOOP

     HERE:
0040|     SJMP HERE

     ORG 0100H
     TABLE:
        DB 11H, 22H, 33H, 44H, 55H

     END
```

What this code does:

1. Sets DPTR to point to a table in code memory
2. Uses MOVC to read each byte from the table
3. Displays each value on Port 1 (P1)
4. Demonstrates that code memory can store constant data

Screenshots of the first module:

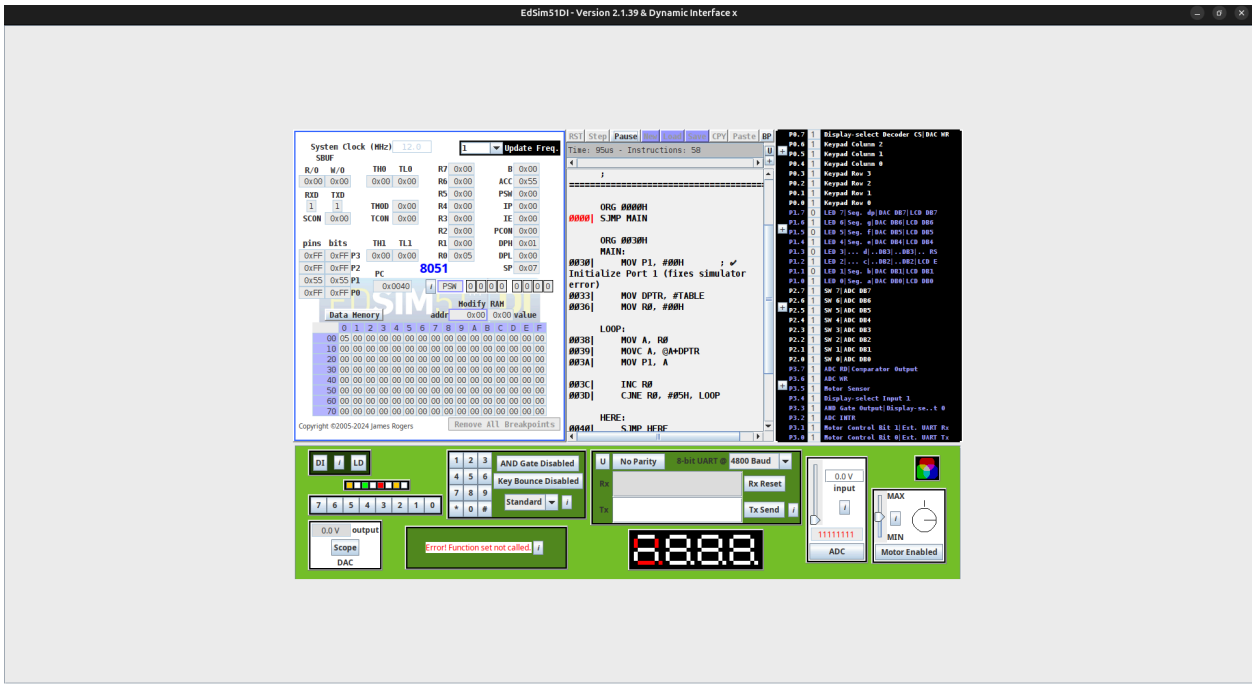
Here I'm showing the simulator **before execution starts**.
The program has already been assembled and loaded into **code memory**, but no instructions have been executed yet.
At this point, **Port 1 is still in its default state**, and the **data memory (RAM) hasn't changed**, which confirms the system is at reset/idle state.



After running the program, you can see the system looping through instructions.

The processor is using the `MOVC` instruction to read values directly from **code memory**, then sending those values to **Port 1**.

The changing output confirms the lookup table in program memory is being accessed correctly.

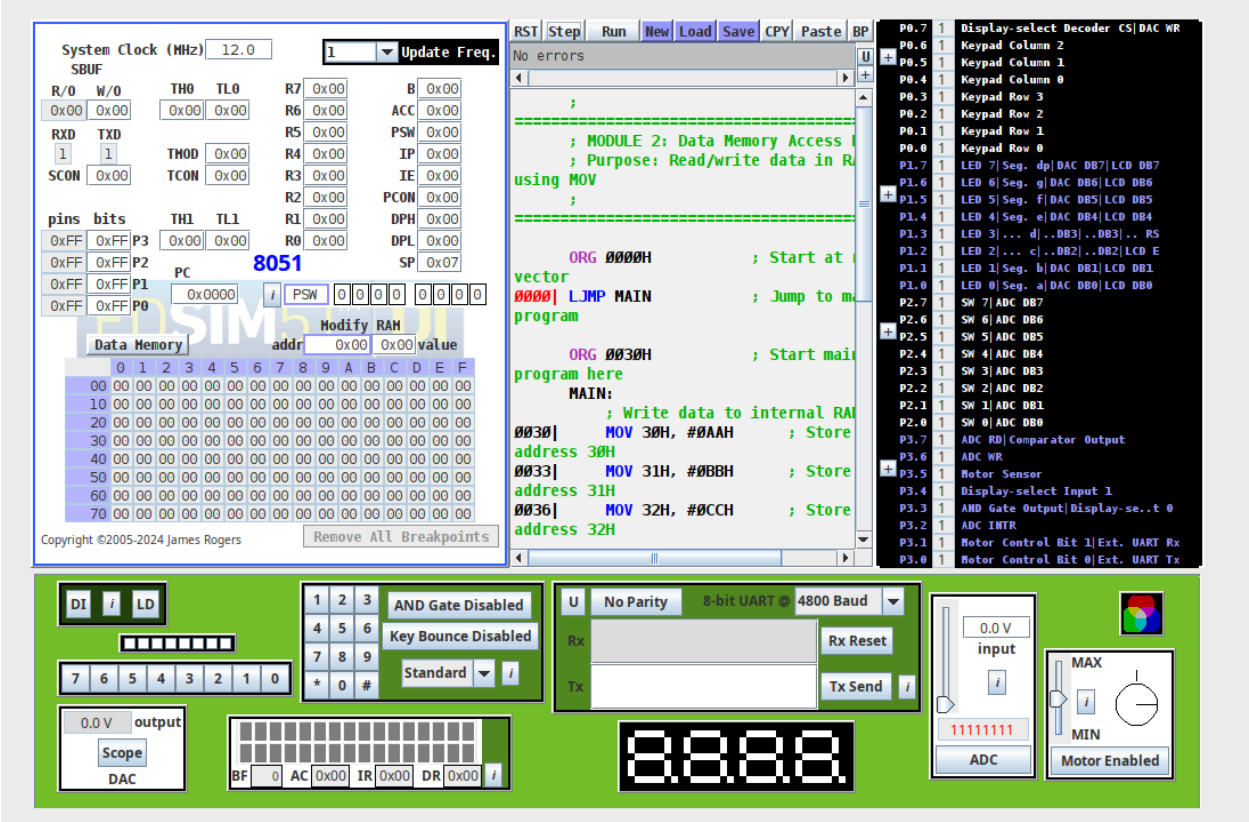


MODULE 2: Data Memory Access (MOV)

```
; ======================================
; MODULE 2: Data Memory Access Demo
; Purpose: Read/write data in RAM using MOV
; ======================================

ORG 0000H          ; Start at reset vector
LJMP MAIN          ; Jump to main program

ORG 0030H          ; Start main program here
MAIN:
   ; Write data to internal RAM
   MOV 30H, #0AAH      ; Store AAH at address 30H
   MOV 31H, #0BBH      ; Store BBH at address 31H
   MOV 32H, #0CCH      ; Store CCH at address 32H

   ; Read data from RAM using direct addressing
   MOV A, 30H          ; Load AAH into accumulator
   MOV P1, A           ; Display on Port 1

   ; Read using indirect addressing
   MOV R0, #31H        ; R0 points to address 31H
   MOV A, @R0          ; Read data at address pointed by R0
   MOV P2, A           ; Display on Port 2

   ; Modify data in RAM
   MOV A, 32H          ; Read CCH
   ADD A, #11H         ; Add 11H (result = DDH)
   MOV 33H, A          ; Store result at 33H

   SJMP $              ; Halt (infinite loop)

END
```

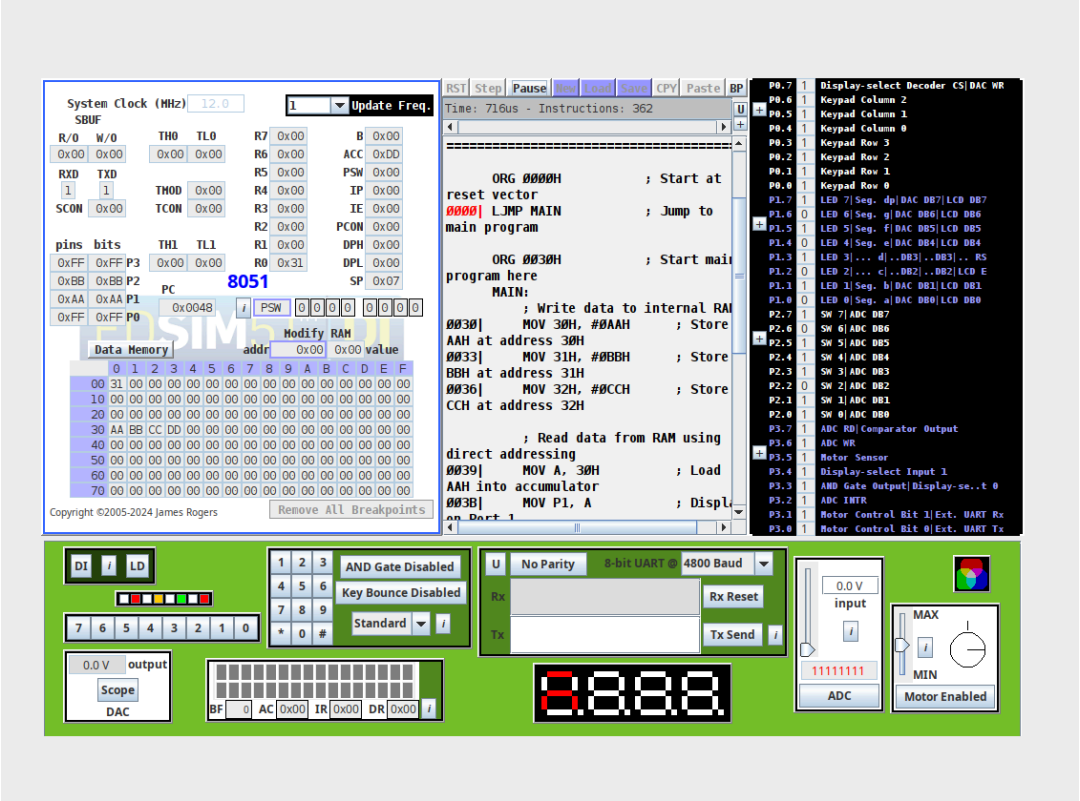
Screenshots for the second module:

At this point, the program has been **assembled and loaded but not executed**.
I'm looking at the **internal data memory (RAM)**, which still contains default values, meaning no write operations have occurred yet.
The accumulator and general-purpose registers are in their initial state, and **Ports 1 and 2 have not been updated**, confirming the system is at a clean starting point before RAM access begins.



Here I'm showing the system **after the program has executed the RAM operations**.
The values `AAH`, `BBH`, and `CCH` have been written into **internal RAM addresses 30H, 31H, and 32H**, which you can see clearly in the **Data Memory window**.
The accumulator is loaded directly from RAM using the `MOV` instruction, and the value from address `30H` is sent to **Port 1**, while the value from address `31H` is accessed indirectly through register `R0` and sent to **Port 2**.

This confirms correct **read and write operations in data memory** using both **direct and indirect addressing**.



What this code does:

1. Writes data to internal RAM locations
2. Reads data using both direct and indirect addressing
3. Demonstrates arithmetic operations on RAM data
4. Shows that data memory is read/write

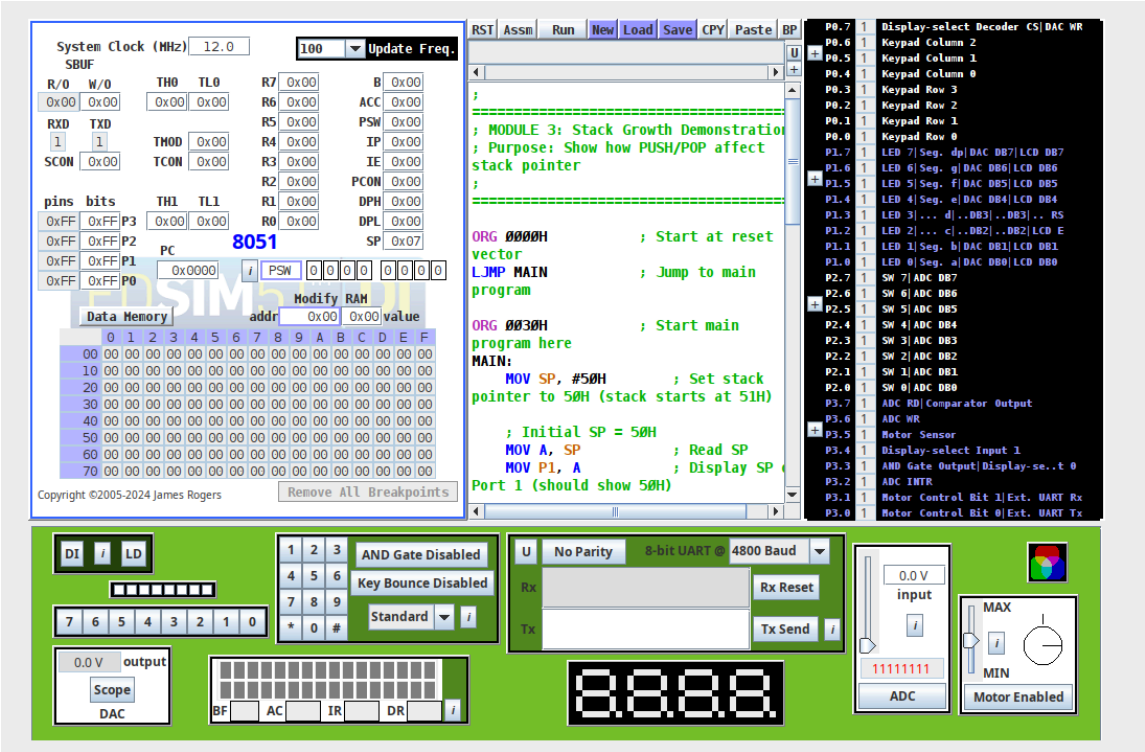
MODULE 3: Stack Growth Demo

```
; ======================================
; MODULE 3: Stack Growth Demonstration
; Purpose: Show how PUSH/POP affect stack pointer
; ======================================

ORG 0000H          ; Start at reset vector
LJMP MAIN          ; Jump to main program

ORG 0030H          ; Start main program here
MAIN:
   MOV SP, #50H       ; Set stack pointer to 50H (stack starts at 51H)

   ; Initial SP = 50H
   MOV A, SP          ; Read SP
   MOV P1, A          ; Display SP on Port 1 (should show 50H)

   ; Push operations (stack grows upward)
   MOV A, #0AAH
   PUSH ACC           ; Push A onto stack (SP = 51H, [51H] = AAH)

   MOV A, SP
   MOV P1, A          ; Display SP (should show 51H)

   MOV B, #0BBH
   PUSH B             ; Push B onto stack (SP = 52H, [52H] = BBH)

   MOV A, SP
   MOV P1, A          ; Display SP (should show 52H)

   MOV R7, #0CCH
   PUSH 07H           ; Push R7 onto stack (SP = 53H, [53H] = CCH)

   MOV A, SP
   MOV P1, A          ; Display SP (should show 53H)

   ; Pop operations (stack shrinks downward)
   POP 06H            ; Pop into R6 (SP = 52H, R6 = CCH)

   MOV A, SP
   MOV P1, A          ; Display SP (should show 52H)
```

```
    POP B              ; Pop into B (SP = 51H, B = BBH)
    POP ACC            ; Pop into A (SP = 50H, A = AAH)

    MOV A, SP
    MOV P1, A          ; Display SP (should show 50H - back to original)

    SJMP $             ; Halt

END
```

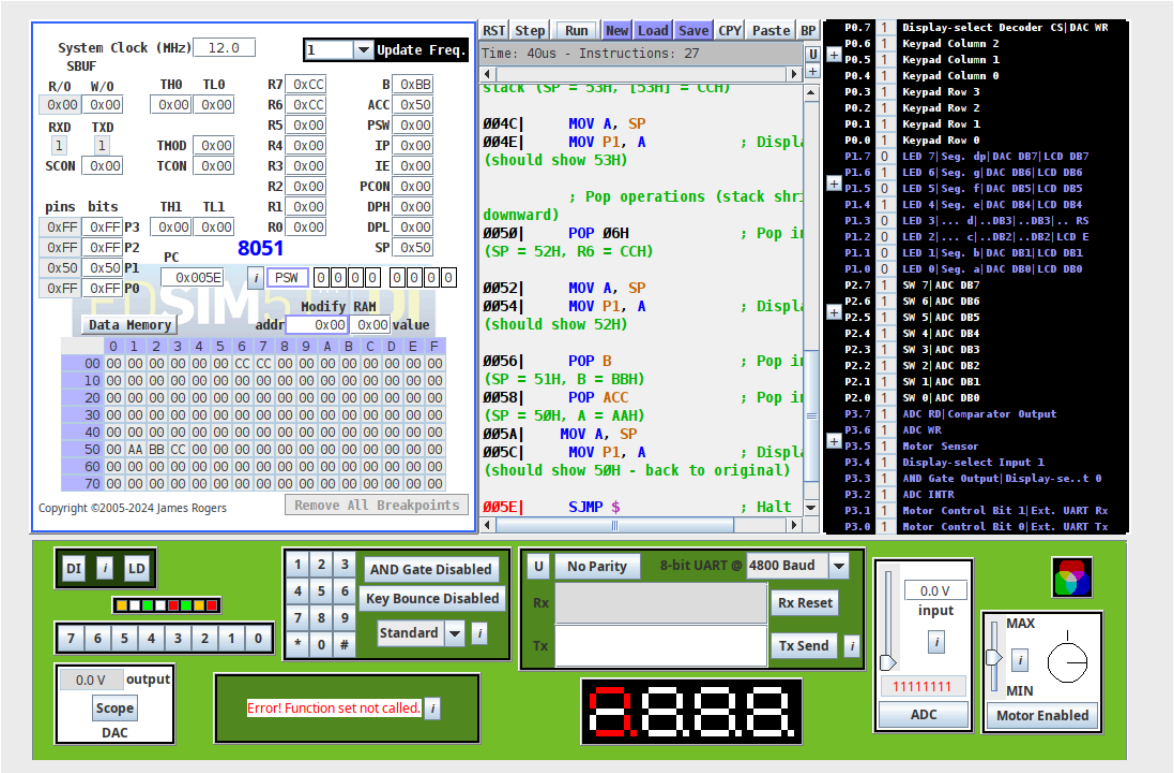
Screenshots for the third module:

This screenshot shows the system **before executing the stack operations**.
The stack pointer has not yet been moved, and **no PUSH or POP instructions have been executed**.
Internal RAM locations above the stack area are unchanged, and **Port 1 has not yet been updated**, confirming that the stack is in its initial, empty state before growth begins.



After executing the program, this screenshot shows **how the stack grows and shrinks in RAM**.
The stack pointer is first set to `50H` and then **increments with each PUSH operation**, showing the stack growing upward as values are stored.
During the POP operations, the stack pointer **decrements back to its original value**, confirming correct stack behavior.

The changing values displayed on **Port 1** reflect the current stack pointer at each stage of execution.



What this code does:

1. Initializes the Stack Pointer to 50H
2. Pushes three values onto the stack and shows SP incrementing
3. Pops three values off the stack and shows SP decrementing
4. Displays SP value on Port 1 after each operation

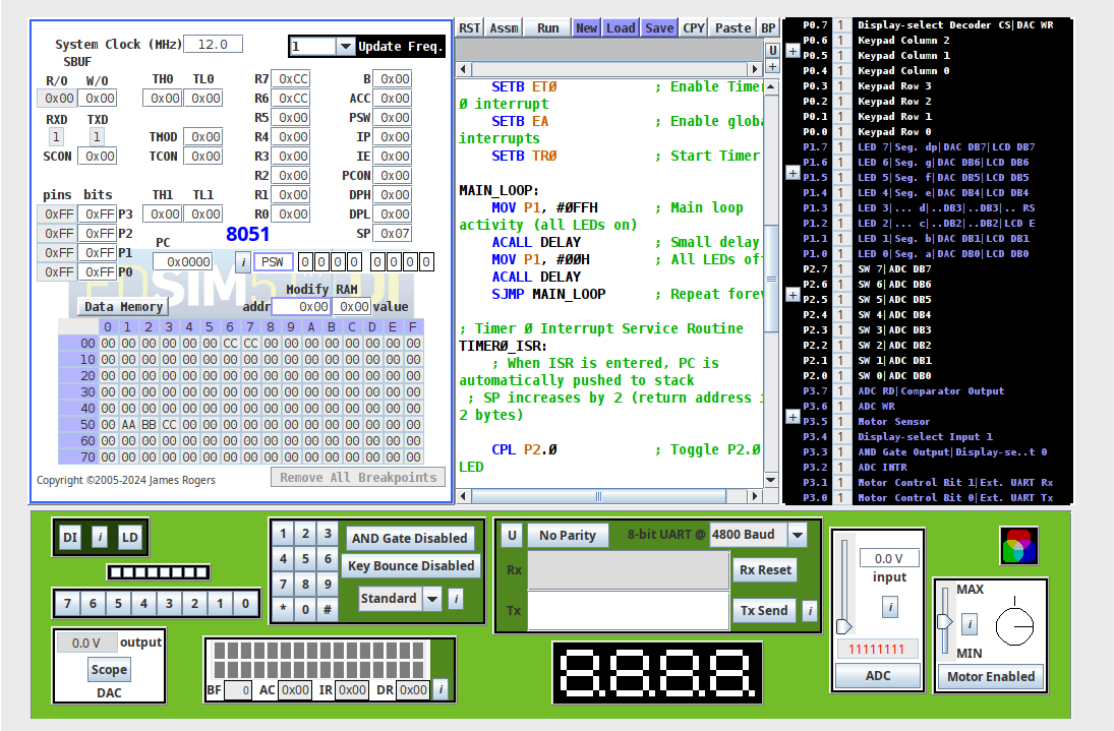
MODULE 4: ISR Stack Usage

```
; ======================================
; MODULE 4: ISR Stack Usage Demo
; Purpose: Show automatic stack usage during interrupts
; ======================================

ORG 0000H          ; Reset vector
LJMP MAIN          ; Jump to main program

ORG 000BH          ; Timer 0 interrupt vector
LJMP TIMER0_ISR    ; Jump to ISR

ORG 0030H          ; Main program
MAIN:
   MOV SP, #60H       ; Set stack pointer to 60H

   ; Configure Timer 0
   MOV TMOD, #01H     ; Timer 0, Mode 1 (16-bit timer)
   MOV TH0, #0FCH     ; Load high byte for short delay
   MOV TL0, #018H     ; Load low byte

   SETB ET0           ; Enable Timer 0 interrupt
   SETB EA            ; Enable global interrupts
   SETB TR0           ; Start Timer 0

MAIN_LOOP:
   MOV P1, #0FFH      ; Main loop activity (all LEDs on)
   ACALL DELAY        ; Small delay
   MOV P1, #00H       ; All LEDs off
   ACALL DELAY
   SJMP MAIN_LOOP     ; Repeat forever

; Timer 0 Interrupt Service Routine
TIMER0_ISR:
   ; When ISR is entered, PC is automatically pushed to stack
   ; SP increases by 2 (return address is 2 bytes)
```

```
    CPL P2.0          ; Toggle P2.0 LED

    ; Reload timer values
    MOV TH0, #0FCH
    MOV TL0, #018H

    RETI              ; Return from interrupt (PC auto-popped)

; Simple delay subroutine
DELAY:
    PUSH 00H          ; Save R0 (SP increases)
    PUSH 01H          ; Save R1 (SP increases)

    MOV R0, #05H
DEL1:
    MOV R1, #0FFH
DEL2:
    DJNZ R1, DEL2
    DJNZ R0, DEL1

    POP 01H           ; Restore R1 (SP decreases)
    POP 00H           ; Restore R0 (SP decreases)
    RET               ; Return from subroutine (PC popped)

END
```

Screenshots for module 4:

This screenshot shows the system **before enabling interrupts and starting Timer 0**.
The stack pointer has just been set to `60H`, and **no interrupt activity has occurred yet**.
Internal RAM above the stack area is unchanged, and **Port 2.0 has not toggled**, indicating that the interrupt service routine has not been entered.
At this stage, the stack contains no return addresses or saved registers.

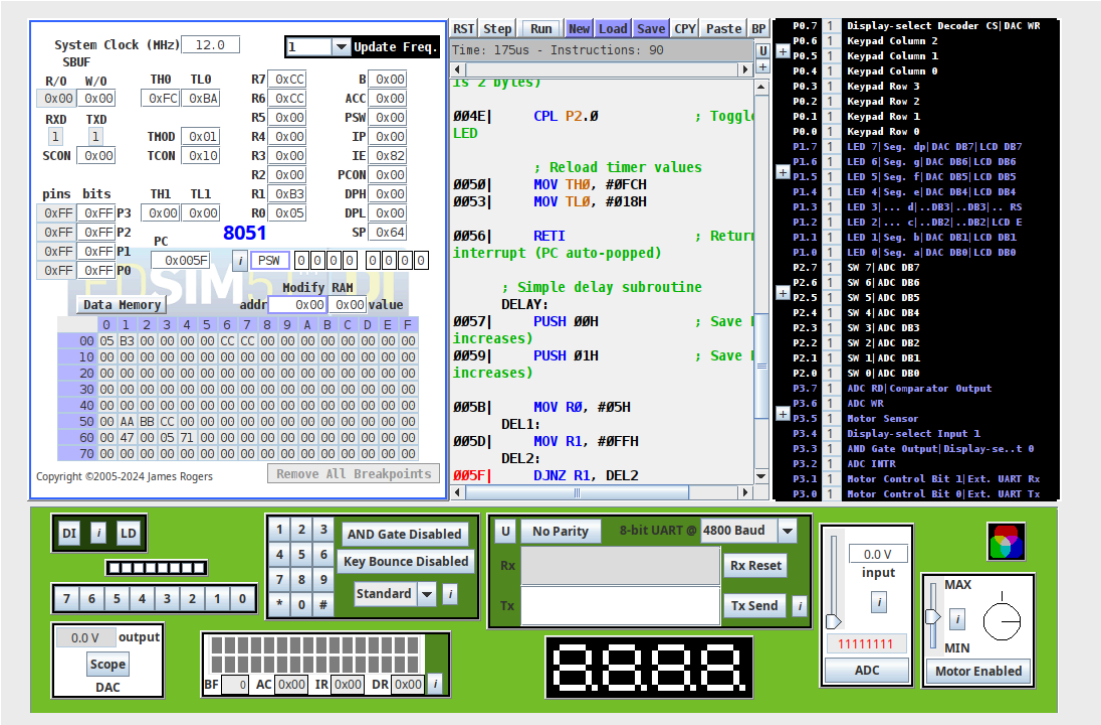


After the program starts running, Timer 0 generates periodic interrupts.

Each time the interrupt occurs, the processor automatically pushes the program counter onto the stack, increasing the stack pointer by two bytes.

Inside the interrupt routine and delay subroutine, additional PUSH instructions further increase stack usage.

When the ISR and subroutines complete, the stack pointer returns to its previous value, confirming correct automatic and manual stack management during interrupt execution.



What this code does:

1. Configures Timer 0 to generate periodic interrupts
2. Main loop toggles Port 1 LEDs
3. When interrupt occurs, the ISR toggles a Port 2 LED
4. Demonstrates automatic stack usage: PC is pushed when entering ISR, popped when exiting
5. Also shows manual stack usage in the DELAY subroutine

7. ANALYSIS

9.1 Comparison of Memory Access Methods

Table 1: Code Memory vs Data Memory

Characteristic	Code Memory	Data Memory
Purpose	Stores program instructions and constants	Stores variables and temporary data
Typical Type	ROM, EPROM, Flash	RAM (SRAM)
Access Speed	Slower (non-volatile memory)	Faster (volatile memory)
Write Capability	Read-only during execution	Read/Write
8051 Instruction	`MOVC` (Move Code)	`MOV`, `MOVX`
Address Range	0000H-FFFFH (64KB)	Internal: 00H-7FH; External: 0000H-FFFFH
Typical Use	Lookup tables, strings, calibration data	Runtime variables, buffers, calculations

Table 2: Harvard vs Von Neumann Architecture

Feature	Harvard Architecture	Von Neumann Architecture
Memory Organization	Separate code and data memory	Unified memory for code and data
Buses	Separate buses for instruction and data	Single shared bus
Simultaneous Access	Yes - can fetch instruction and access data simultaneously	No - bus contention between instruction fetch and data access
Performance	Higher (no bus bottleneck)	Lower (Von Neumann bottleneck)
Complexity	More complex (dual buses)	Simpler (single bus)

Security	Better (code protected from data overwrites)	Lower (accidental code modification possible)
Examples	8051, PIC, AVR, ARM Cortex-M (modified)	Early Intel 8086, x86 computers

Table 3: Stack Behavior Summary

Operation	Effect on SP	Memory Access	Use Case
PUSH	SP = SP + 1, then store	Writes to [SP]	Save register values, pass parameters
POP	Read from [SP], then SP = SP - 1	Reads from [SP]	Restore register values
CALL/ACALL	Auto-push return address (2 bytes)	Writes PC to stack	Subroutine call
RET	Auto-pop return address	Reads PC from stack	Return from subroutine
ISR Entry	Auto-push PC (2 bytes)	Writes PC to stack	Hardware interrupt
RETI	Auto-pop PC	Reads PC from stack	Return from interrupt

9.2 Key Observations

1. **Code vs Data Memory Access:**
    - The `MOVC` instruction is specifically designed for reading from code memory and requires DPTR or PC as the base address
    - The `MOV` instruction accesses data memory (internal RAM) and supports multiple addressing modes
    - This separation enforces the Harvard architecture principle
2. **Stack Growth Pattern:**
    - The 8051 stack grows **upward** (toward higher memory addresses)
    - Each PUSH increments SP **before** storing data
    - Each POP retrieves data **before** decrementing SP

    - This is opposite to some architectures (like x86) where the stack grows downward
3. **ISR Stack Usage:**

    - Interrupt handling automatically uses the stack without programmer intervention
    - The return address (Program Counter) is 2 bytes on the 8051
    - Proper stack management is critical - stack overflow can corrupt data
    - The SP should be initialized to a safe location (typically 60H or higher) to avoid overwriting register banks
4. **Harvard Architecture Benefits Observed:**

    - Code memory remains protected during execution
    - Different instructions for different memory types make programming more explicit
    - Lookup tables in code memory save valuable RAM space
1. **Memory Initialization:**

    - Uninitialized RAM contains random values
    - Good practice: always initialize variables before use
    - Our code explicitly initialized all RAM locations before reading them
    - Hardware revision should be noted for physical implementations

8. CONCLUSION

This laboratory successfully demonstrated the fundamental differences between Von Neumann and Harvard architectures through hands-on simulation of the 8051 microcontroller. By implementing and observing four distinct assembly programs, we gained practical insight into memory organization and access methods.

Key Learnings:

1. **Architecture Understanding:** The 8051's Harvard architecture provides clear separation between code and data memory, offering both performance benefits and code protection.

2. **Memory Access Methods:** Different instructions (`MOVC` vs `MOV`) are required for different memory types, making the architecture's design explicit in the code.

3. **Stack Mechanics:** The stack is a dynamic data structure that grows upward in the 8051, and it's essential for both subroutine calls and interrupt handling.

4. **ISR Behavior:** Interrupts automatically manage the stack to save and restore execution context, but programmers must ensure adequate stack space is available.

5. **Practical Implications:** Understanding memory architecture is essential for:

    - Writing efficient code
    - Debugging memory-related issues
    - Preventing stack overflow
    - Optimizing RAM usage by storing constants in code memory