Mpmc (4341101) - Problem Solver

Antigravity AI

June 4, 2026

Mpmc

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Contents

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

CHAPTER 1

Introduction to Microprocessor

In this chapter, we focus on the fundamental computational aspects of microprocessors. While introductory topics often cover theoretical architecture, mastering the numerical analysis of memory sizes, address bus mapping, and execution timing is essential for solving concrete hardware interfacing problems.

1.1 Memory Capacity and Address Lines

A microprocessor communicates with its memory via an address bus. The number of parallel lines in this address bus strictly dictates the maximum amount of physical memory the microprocessor can directly access.

Methodology:

Calculating Memory Capacity from Address Lines To find the maximum memory capacity (M) that can be addressed by a microprocessor having an address bus with N lines:

1. Identify the total number of address lines, N.
2. The total number of unique memory locations the processor can point to is given by 2^N.
3. Assuming standard byte-addressable memory (where each location stores 1 Byte), the total memory capacity is $M = 2^N$ Bytes.
4. Convert the result into more readable units (KB, MB, GB) using the base-2 unit conventions:
   - 1 KB $= 2^{10}$ Bytes $= 1024$ Bytes
   - 1 MB $= 2^{20}$ Bytes $= 1024$ KB $= 1,048,576$ Bytes
   - 1 GB $= 2^{30}$ Bytes $= 1024$ MB

Worked Example:

Memory Capacity of an 8085 Microprocessor The classical 8085 microprocessor utilizes a 16-bit address bus. Calculate the maximum memory capacity it can natively access.
Solution:

1. The number of address lines is given as $N = 16$.
2. Calculate the total possible unique memory locations:

$$\text{Locations} = 2^{16}$$

3. Assuming byte-addressable memory, the total capacity is 2^{16} Bytes.
4. To convert this raw byte count into Kilobytes (KB), factor out 2^{10}:

$$2^{16} = 2^6 \times 2^{10} \text{ Bytes}$$

5. Substitute the values:

$$2^6 = 64$$

$$2^{10} \text{ Bytes} = 1 \text{ KB}$$

$$\text{Capacity} = 64 \times 1 \text{ KB} = 64 \text{ KB}$$

The 8085 microprocessor can address a maximum of 64 KB of memory.

Methodology:

Calculating Required Address Lines for a Specific Memory Size Conversely, to determine the minimum number of address pins (N) a memory chip must have to expose its entire capacity (M) to the system bus:

1. Obtain the memory capacity M and convert it entirely into Bytes.
2. Express the total byte count as a power of 2, setting up the equation $M = 2^N$.
3. Solve for the exponent N. This value represents the required number of address lines.

Worked Example:

Address Lines for a 256 KB Memory Module Determine the number of address lines required to interface a 256 KB SRAM memory module.
Solution:

1. The given memory capacity is $M = 256$ KB.
2. Convert this value into Bytes:

$$M = 256 \times 1024 \text{ Bytes} = 256 \times 2^{10} \text{ Bytes}$$

3. Express the scalar 256 as a base-2 power: $256 = 2^8$.
4. Multiply the powers to find the total capacity in base-2:

$$M = 2^8 \times 2^{10} = 2^{(8+10)} = 2^{18} \text{ Bytes}$$

5. Equate to the capacity formula $2^N = 2^{18}$, yielding $N = 18$.

The 256 KB memory module requires exactly 18 address lines.

1.2 Memory Mapping and Address Ranges

Memory mapping establishes the exact hexadecimal bounds (start and end addresses) assigned to a memory chip within the processor's overall memory space.

Methodology:

Finding the Memory Address Range Given a starting base address and the capacity of the memory chip, you can determine its ending address:

1. Convert the given memory chip capacity into total Bytes (locations). Let this be L.
2. Convert the decimal value of L into hexadecimal notation.
3. Compute the address offset (Δ) by subtracting 1 from the total locations.

$$\Delta = L_{hex} - 1H$$

4. Add the starting address (A_{start}) to the calculated address offset (Δ) to find the ending address (A_{end}).

$$A_{end} = A_{start} + \Delta$$

Worked Example:

Address Range Calculation for a 4 KB Memory Chip A 4 KB memory chip is interfaced with a 16-bit microprocessor. If its decoding logic sets its starting address at $2000H$, calculate its ending address.
Solution:

1. Determine the total locations L for a 4 KB capacity:

$$4 \text{ KB} = 4 \times 1024 = 4096 \text{ Bytes}$$

2. Convert the decimal value 4096 to hexadecimal. We know $4096 = 16^3$, which translates to $1000H$.

$$L = 1000H$$

3. Calculate the address offset Δ:

$$\Delta = L - 1H = 1000H - 0001H = 0FFFH$$

4. Calculate the ending address by adding the offset to the starting address:

$$A_{end} = 2000H + 0FFFH = 2FFFH$$

The address range assigned to the 4 KB memory chip spans from $2000H$ to $2FFFH$.

1.3 Instruction Execution Timing Analysis

The time required by a microprocessor to fetch and execute an instruction relies on its hardware clock frequency and the internal number of clock cycles (T-states) dictated by the instruction's microcode.

Methodology:

Calculating Instruction Execution Time To accurately compute the total execution time (T_{exec}) of a specific instruction:

1. Identify the processor's operating clock frequency (f).
2. Calculate the duration of a single clock cycle (T-state), $T = \frac{1}{f}$.
3. Look up or determine the total number of T-states required by the instruction (N_T).
4. Multiply the duration of one T-state by the total number of T-states:

$$T_{exec} = N_T \times T = \frac{N_T}{f}$$

Worked Example:

Execution Time of an Immediate Load Instruction A microprocessor operates at a clock frequency of 3 MHz. An `MVI A, 45H` instruction requires 7 T-states to execute. Calculate the absolute execution time of this instruction.
Solution:

1. The operating clock frequency is $f = 3 \text{ MHz} = 3 \times 10^6 \text{ Hz}$.

2. Determine the duration of a single T-state:

$$T = \frac{1}{f} = \frac{1}{3 \times 10^6} \text{ seconds} \approx 0.333 \times 10^{-6} \text{ seconds} = 0.333\,\mu\text{s}$$

3. Total number of T-states for the instruction is $N_T = 7$.

4. Compute the total execution time:

$$T_{exec} = 7 \times 0.333\,\mu\text{s} = 2.331\,\mu\text{s}$$

The instruction completely executes in 2.331 μs.

Worked Example:

Delay Calculation for a Software Loop Software delay loops generate wait times by repeatedly executing instructions. A particular loop consists of instructions taking a cumulative 24 T-states per iteration. If the loop register is initialized to FFH (255 decimal) and the processor runs at 2 MHz, calculate the total time delay generated by the loop.

Solution:

1. The operating clock frequency is $f = 2$ MHz $= 2 \times 10^6$ Hz.

2. Determine the duration of one T-state:

$$T = \frac{1}{2 \times 10^6} = 0.5 \times 10^{-6} \text{ seconds} = 0.5\,\mu\text{s}$$

3. Total T-states per single iteration of the loop is 24.

4. Since the loop runs 255 times, calculate the total T-states executed over the entire loop lifecycle:

$$\text{Total } N_T = 24 \times 255 = 6120 \text{ T-states}$$

5. Multiply by the duration of a single T-state to find the total time delay:

$$T_{delay} = 6120 \times 0.5\,\mu\text{s} = 3060\,\mu\text{s}$$

6. Convert to milliseconds for better readability:

$$3060\,\mu\text{s} = 3.06 \text{ ms}$$

The software loop accurately generates a delay of 3.06 ms.

1.4 Memory Interfacing and Chip Sizing

When engineering a memory subsystem, engineers often need to gang multiple smaller memory chips together to satisfy a requirement for a larger overall memory capacity or a wider data bus.

Methodology:

Calculating the Number of Required Memory Chips To determine the quantity of smaller memory ICs needed to construct a target memory system:

1. Identify the total capacity requirements of the target system (C_{target}).

2. Identify the individual capacity of the available memory chip component (C_{chip}).

3. Ensure both capacities are represented in matching units (e.g., total bits, total Bytes, or words × bits).
4. The total number of required chips (N_{chips}) is calculated by standard division:

$$N_{chips} = \frac{C_{target}}{C_{chip}}$$

Worked Example:

Designing a Memory System with Smaller ICs A system requires a 16 KB memory block, but only 4 KB memory chips are currently in inventory. How many individual chips must be cascaded?
Solution:

1. Target overall memory capacity $C_{target} = 16$ KB.
2. Individual chip capacity $C_{chip} = 4$ KB.
3. Since both are expressed in Kilobytes, calculate the required number directly:

$$N_{chips} = \frac{16 \text{ KB}}{4 \text{ KB}} = 4$$

A total of 4 independent memory chips (each 4 KB) must be connected in series to map out the full 16 KB memory system.

Worked Example:

Expanding Memory Word Size Parallelization A microprocessor natively utilizes an 8-bit data bus and requires a 2048 × 8-bit memory grid. The only available memory components on the market are 2048 × 4-bit chips. Calculate the structural layout and number of chips required.
Solution:

1. The target memory system size is exactly 2048 words × 8 bits/word.
2. The available chip footprint is 2048 words × 4 bits/word.
3. First, divide the required number of words by the chip's word capacity to find how many rows of chips are needed:

$$\text{Number of chip rows} = \frac{2048 \text{ words}}{2048 \text{ words}} = 1 \text{ row}$$

4. Next, divide the target data bus width by the chip's data pin count to see how many chips must sit in parallel to output the necessary width:

$$\text{Number of chip columns} = \frac{8 \text{ bits}}{4 \text{ bits}} = 2 \text{ parallel chips}$$

5. The total number of memory chips required is the product of rows and columns:

$$N_{chips} = 1 \text{ row} \times 2 \text{ parallel chips} = 2 \text{ total chips}$$

The engineering team must deploy 2 memory chips wired in parallel. This configuration shares the address lines while distributing the 8-bit data bus split across both chips (4 bits each).

CHAPTER 2

Working of 8085 Microprocessor

The 8085 microprocessor is an 8-bit general-purpose processor. Understanding its architecture requires analyzing its internal registers, multiplexed bus structures, and the execution sequence of its instructions. This chapter focuses on the computational aspects of processor state management and execution timing.

2.1 Register Organization and Flag Operations

The 8085 contains an 8-bit Accumulator (Register A), six 8-bit general-purpose registers (B, C, D, E, H, L), a 16-bit Program Counter (PC), and a 16-bit Stack Pointer (SP). Evaluating the processor state involves tracing the contents of these registers and the status flags after operations.

Methodology:

Flag Register Evaluation The 8085 has a 5-bit flag register that reflects the outcome of arithmetic and logical operations. The 8-bit register format is `S | Z | X | AC | X | P | X | CY`, where `X` bits are undefined (typically 0).

1. **Sign (S):** Set to 1 if the Most Significant Bit (MSB) of the result is 1 (indicating a negative value in 2's complement), otherwise 0.
2. **Zero (Z):** Set to 1 if the entire 8-bit result is exactly zero, otherwise 0.
3. **Auxiliary Carry (AC):** Set to 1 if a carry is generated from bit 3 to bit 4 during addition, otherwise 0.
4. **Parity (P):** Set to 1 if the binary result contains an even number of 1s (even parity), otherwise 0.
5. **Carry (CY):** Set to 1 if a carry is generated out of the MSB (bit 7) during addition, or a borrow occurs during subtraction, otherwise 0.

Worked Example:

Evaluate Flag Register after Addition Given that the Accumulator contains 0x85 and the B register contains 0x3C. Calculate the value of the Accumulator and the Flag register after the instruction `ADD B` is executed.
Solution:

1. Convert the hexadecimal values to binary:

$$A(\text{0x85}) = 1000\ 0101_2$$
$$B(\text{0x3C}) = 0011\ 1100_2$$

2. Perform the binary addition:

$$\begin{array}{rr} & 1000\ 0101 \\ + & 0011\ 1100 \\ \hline & 1100\ 0001 \end{array}$$

The result stored in the Accumulator is $1100\ 0001_2 = 0xC1$.

3. Evaluate the Flags based on the result $1100\ 0001_2$:
   - **Sign (S):** The MSB (bit 7) is 1, so $S = 1$.
   - **Zero (Z):** The result is not entirely zero, so $Z = 0$.
   - **Auxiliary Carry (AC):** Addition of the lower nibbles $(0101 + 1100)$ produces a sum of 10001. A carry moves from bit 3 to bit 4, so $AC = 1$.
   - **Parity (P):** The result has exactly three 1s (odd parity), so $P = 0$.
   - **Carry (CY):** No carry out of bit 7 occurred, so $CY = 0$.
4. Construct the Flag register byte:

S	Z	X	AC	X	P	X	CY
1	0	0	1	0	0	0	0

The final Flag register content is $1001\ 0000_2 = 0x90$.

Methodology:

Program Counter and Stack Pointer Updates The Program Counter (PC) tracks the 16-bit address of the next instruction byte to fetch. The Stack Pointer (SP) points to the top of the stack in memory.

1. **PC Increment:** For every byte of the instruction fetched from memory, the PC increments by 1. For a 3-byte instruction, the PC increments three times during the fetch phase.
2. **SP Update on PUSH:** When pushing a 16-bit value onto the stack, the SP decrements by 1, the high-order byte is stored, the SP decrements by 1 again, and the low-order byte is stored. Overall, SP decreases by 2.
3. **SP Update on POP:** When popping data, the low-order byte is retrieved, SP increments by 1, the high-order byte is retrieved, and SP increments by 1 again. Overall, SP increases by 2.

Worked Example:

Calculating PC and SP Values An 8085 program is executing. The current instruction, `CALL 2050H` (a 3-byte instruction), is located at memory address `8000H`. The Stack Pointer (SP) initially contains `C000H`. Determine the values of the PC and SP after the `CALL` instruction is completely executed.
Solution:

1. **Determine the Return Address:** The `CALL` instruction occupies 3 bytes starting at `8000H` (stored at `8000H`, `8001H`, and `8002H`). The next sequential instruction address is `8003H`.
2. **Update PC for Fetch Phase:** After fetching the 3 bytes, the PC naturally points to the return address: PC = 8003H.
3. **Execute CALL:** The `CALL` instruction pushes the 16-bit return address (`8003H`) onto the stack.
   - Decrement SP: SP = C000H − 1 = BFFFH.
   - Store high-order byte (`80H`) at `BFFFH`.
   - Decrement SP: SP = BFFFH − 1 = BFFEH.
   - Store low-order byte (`03H`) at `BFFEH`.
4. **Update PC for Branching:** The PC is loaded with the target address of the `CALL`. So, the new PC value is PC = 2050H.
5. **Final Values:** After execution completes, PC = 2050H and SP = BFFEH.

2.2 Bus Structure and Demultiplexing

To reduce the pin count, the 8085 multiplexes the lower 8 bits of the address bus (A_0-A_7) with the 8-bit data bus (D_0-D_7) to form the AD_0-AD_7 pins. The upper 8 bits of the address bus (A_8-A_{15}) remain dedicated.

Methodology:

Demultiplexing Address and Data Bus The Address Latch Enable (ALE) signal separates the address and data information on the AD_0-AD_7 lines.

1. Identify the machine cycle state. During the first clock cycle (T_1) of any machine cycle, the microprocessor places the lower 8-bit address on the AD_0-AD_7 lines.
2. The processor asserts the ALE signal (ALE = 1) during T_1. This active-high signal latches the address into an external latch (e.g., 74LS373 IC).
3. During subsequent clock cycles (T_2 and T_3), ALE goes low (ALE = 0), and the AD_0-AD_7 lines transition to data mode for reading or writing.

Worked Example:

Bus State Identification The 8085 is reading data from memory address 2055H. Determine the values on the A_8-A_{15} pins, AD_0-AD_7 pins, and the state of the ALE signal during states T_1 and T_2 of the Memory Read cycle.
Solution:

1. The target memory address is 2055H. Break the address into bytes:
   - High-order byte: 20H
   - Low-order byte: 55H
2. Analyze the T_1 state (Address Phase):
   - The higher-order address is placed on A_8-A_{15}, so A_8-A_{15} = 20H.
   - The lower-order address is placed on AD_0-AD_7, so AD_0-AD_7 = 55H.
   - ALE is active to indicate a valid address. ALE = 1.
3. Analyze the T_2 state (Data Phase):
   - The higher-order address remains on A_8-A_{15}, so A_8-A_{15} = 20H.
   - The AD_0-AD_7 lines switch to data mode to receive the incoming byte. AD_0-AD_7 = Data stored at 2055H.
   - ALE goes inactive. ALE = 0.

Methodology:

Memory Capacity Calculation The address bus width dictates the maximum addressable memory capacity of the processor.

1. Determine the number of unique address lines (n).
2. Calculate total addressable locations using the formula: Locations = 2^n.
3. Multiply by the word size (1 byte for 8085) to yield total capacity.

Worked Example:

Calculating 8085 Memory Capacity Verify the total addressable memory capacity of the 8085 microprocessor.
Solution:

1. The 8085 has a dedicated upper address bus (A_8-A_{15}) of 8 lines and a multiplexed lower address bus (AD_0-AD_7) of 8 lines.
2. Total number of address lines: $n = 8 + 8 = 16$.
3. Calculate the total memory locations:

$$\text{Total Locations} = 2^{16} = 65,536$$

4. Since each memory location stores 1 byte, the total capacity is $65,536$ bytes.
5. Convert to Kilobytes:

$$\text{Capacity in KB} = \frac{65,536}{1024} = 64 \text{ KB}$$

2.3 Instruction Fetching Decoding and Execution

An instruction cycle consists of one or more machine cycles (such as Opcode Fetch, Memory Read, Memory Write). Each machine cycle consists of several T-states (clock cycles).

Methodology:

Execution Time Calculation The overall execution time of an instruction is a function of the required T-states and the system clock frequency.

1. Determine the total number of T-states (N_T) required by summing the T-states of all machine cycles in the instruction.
2. Identify the microprocessor's operating clock frequency (f_{clk}).
3. Calculate the duration of a single T-state, $T = \frac{1}{f_{clk}}$.
4. Compute the total execution time, $t_{exec} = N_T \times T$.

Worked Example:

Calculating Instruction Execution Time An 8085 microprocessor is operating with a clock frequency of 3 MHz. Calculate the execution time for the `MVI A, 32H` instruction. This instruction requires 2 machine cycles (Opcode Fetch requiring 4 T-states, and Memory Read requiring 3 T-states).
Solution:

1. Calculate the total T-states: $N_T = 4 + 3 = 7$.
2. Identify the clock frequency: $f_{clk} = 3 \text{ MHz} = 3 \times 10^6 \text{ Hz}$.
3. Calculate the time duration for one T-state:

$$T = \frac{1}{3 \times 10^6} \text{ seconds} \approx 0.333 \ \mu\text{s}$$

4. Calculate the total execution time:

$$t_{exec} = 7 \times 0.333 \ \mu\text{s} = 2.331 \ \mu\text{s}$$

The instruction takes exactly 2.333 μs (or $\frac{7}{3}\mu$s) to execute.

Worked Example:

Determining Control Signals Determine the status of the $IO/\overline{M}$, S_1, and S_0 control signals during the execution of the `MOV A, M` instruction, which involves an Opcode Fetch cycle and a Memory Read cycle.
Solution:

1. The $IO/\overline{M}$ signal distinguishes between I/O and memory operations. S_1 and S_0 denote the machine cycle type.
2. **Machine Cycle 1: Opcode Fetch**
   - Fetching an opcode is a memory access: $IO/\overline{M} = 0$.
   - For an opcode fetch, status signals are high: $S_1 = 1, S_0 = 1$.
3. **Machine Cycle 2: Memory Read**
   - Reading data from memory (`M`): $IO/\overline{M} = 0$.
   - For a memory read operation, status signals are: $S_1 = 1, S_0 = 0$.

2.4 Comparison of Microprocessor and Microcontroller

Selecting between a microprocessor and a microcontroller relies on analyzing the computational and peripheral requirements of a given problem.

Methodology:

Processor Selection Criteria Apply the following rules to classify an application's processor requirement:

1. **Memory Footprint:** If an application requires gigabytes of external memory, select a microprocessor. If kilobyte-range on-chip memory suffices, select a microcontroller.
2. **Peripheral Integration:** If the design requires built-in ADCs, timers, and specific communication ports (like I^2C or SPI) in a compact layout, choose a microcontroller.
3. **Computational Throughput:** If the system involves heavy, complex math algorithms running at high clock speeds, a microprocessor is mandatory.
4. **System Cost:** Microcontrollers minimize printed circuit board (PCB) size and component count, optimizing low-cost embedded systems.

Worked Example:

Selecting the Appropriate Architecture Evaluate two systems: System A is an automated traffic light controller, and System B is an artificial intelligence training server. Determine the appropriate processor type for each.
Solution:

1. **System A: Traffic Light Controller**
   - **Requirements:** Simple timing logic, switching discrete outputs (LEDs), and low processing overhead.
   - **Selection: Microcontroller**.
   - **Justification:** It utilizes integrated hardware timers to manage delays and built-in I/O ports to directly switch the lights. Small memory requirement fits entirely on-chip, minimizing cost and space.
2. **System B: AI Training Server**

- **Requirements:** Dense matrix multiplications, high-speed access to large datasets in external memory, and maximum computational speed.
- **Selection: Microprocessor** (specifically a specialized high-performance CPU/GPU).
- **Justification:** Requires massive external memory support via wide address buses and extreme computational throughput, which microcontrollers cannot provide. Lack of integrated generic I/O is acceptable.

CHAPTER 3

Microcontroller Architecture

3.1 Core Architecture and Program Status Word

The 8051 microcontroller consists of key functional blocks including the ALU, Program Counter (PC), Data Pointer (DPTR), Program Status Word (PSW), internal RAM and ROM, Special Function Registers (SFRs), Timers/Counters, Interrupt control, and I/O Ports. A critical skill is evaluating the state of the Program Status Word (PSW) after arithmetic operations.

Methodology:

Determining PSW Flag Status The PSW is an 8-bit register containing condition flags: CY (Carry), AC (Auxiliary Carry), F0, RS1, RS0 (Register Bank Select), OV (Overflow), -, and P (Parity).

1. **CY (Carry Flag):** Set to 1 if there is a carry out from the MSB (bit 7) during addition or a borrow during subtraction.
2. **AC (Auxiliary Carry Flag):** Set to 1 if there is a carry from bit 3 to bit 4 (lower nibble to upper nibble).
3. **OV (Overflow Flag):** Set to 1 if there is a carry out of bit 6 but not bit 7 or a carry out of bit 7 but not bit 6.
4. **P (Parity Flag):** Set to 1 if the accumulator (A) has an odd number of 1s (to maintain even parity overall).

Worked Example:

Evaluating PSW flags after Addition Given that Accumulator A = `0x38` and Register B = `0x2F`. Calculate the result of A + B and determine the status of the CY AC OV and P flags.
Solution:

1. **Convert to Binary:** A = `0x38` = `0011 1000` B = `0x2F` = `0010 1111`
2. **Perform Addition:**

```
  0011 1000
+ 0010 1111
-----------
  0110 0111  (Result = 0x67)
```

3. **Determine Flags:**
   - **CY:** There is no carry out of bit 7. CY = 0.
   - **AC:** Adding bit 3 (0) and bit 3 (1) gives 1, no carry from bit 3 to bit 4. AC = 0.
   - **OV:** Carry out of bit 6 is 0, carry out of bit 7 is 0. OV = 0.
   - **P:** The result in A is `0110 0111`. Number of 1s is 5 (odd). P = 1.

3.2 I/O Port Configuration and Pin Operations

The 8051 has four 8-bit bidirectional I/O ports (Port 0 to Port 3). The functions of these pins range from providing I/O to essential control signals (ALE, EA, PSEN) and specialized Port 3 features (RXD, TXD, INT0, INT1).

Methodology:

Configuring Port Pins for Input and Output By default, upon reset, all ports are configured as inputs. To manage port directions:

1. **Configuring as Output:** Write a `0` or `1` to the port latch. The pin will drive the state accordingly.
2. **Configuring as Input:** To configure a pin (or port) as an input after it has been used as an output, write a `1` to the pin. This turns off the internal pull-down transistor, allowing external signals to pull the pin low.
3. **Reading Pin Status:** Read the state of the physical pin directly using instructions that target the port register, ensuring a `1` was previously written to it.

Worked Example:

Determining Port Latch Values for Pin Operations Given Port 1 is initially `0xFF`. We want to output the value `0x55` on Port 1, then read the status of Pin P1.3, and finally toggle the state of P1.7. Determine the sequence of operations and resulting register states.
Solution:

1. **Output Value to Port 1:** Write `0x55` to P1 register.

$$\text{P1} = \texttt{0101 0101}_2$$

   Pins are driven: P1.7=0, P1.6=1, P1.5=0, P1.4=1, P1.3=0, P1.2=1, P1.1=0, P1.0=1.

2. **Configure and Read P1.3:** To reliably read an external signal on P1.3, it must first be configured as an input by writing a `1` to it. Set P1.3 to `1`.

$$\text{P1} = \texttt{0101 1101}_2 = \texttt{0x5D}$$

   The physical pin state can now be read reliably.

3. **Toggle P1.7:** The current state of P1.7 in the latch is `0`. To toggle it, complement the specific bit.

$$\text{New P1.7} = \sim 0 = 1$$

$$\text{P1} = \texttt{1101 1101}_2 = \texttt{0xDD}$$

3.3 Stack and Stack Pointer Operations

The stack in the 8051 is a section of internal RAM used to store data temporarily or hold return addresses. The Stack Pointer (SP) is an 8-bit register pointing to the top of the stack.

Methodology:

Tracking Stack Pointer and Memory The 8051 stack grows upwards in memory (from lower to higher addresses). Upon reset, SP is initialized to `0x07`.

1. **PUSH operation:**
   - SP is incremented by 1 ($SP = SP + 1$).

- The data is stored at the new address pointed to by SP.

2. **POP operation:**
   - Data is retrieved from the address currently pointed to by SP.
   - SP is decremented by 1 ($SP = SP - 1$).

Worked Example:

Tracing SP during PUSH and POP Assume the SP is currently `0x09`. The following sequence of stack operations occurs:

```
PUSH 0xE0    (Acc = 0x55 is at 0xE0)
PUSH 0xF0    (B reg = 0xAA is at 0xF0)
POP 0x00     (R0 of bank 0)
```

Determine the value of SP and stack memory contents after each instruction.
Solution:

1. **Initial State:** SP = `0x09`.
2. **After PUSH 0xE0:**
   - SP is incremented: SP = `0x09` + 1 = `0x0A`.
   - Content of Accumulator (`0x55`) is copied to RAM location `0x0A`.
3. **After PUSH 0xF0:**
   - SP is incremented: SP = `0x0A` + 1 = `0x0B`.
   - Content of B register (`0xAA`) is copied to RAM location `0x0B`.
4. **After POP 0x00:**
   - Data at RAM location `0x0B` (`0xAA`) is copied to `0x00` (R0).
   - SP is decremented: SP = `0x0B` - 1 = `0x0A`.

   Final State: SP = `0x0A`. RAM location `0x0A` contains `0x55`. Register R0 contains `0xAA`.

3.4 Timer and Counter Operations

The 8051 has two 16-bit timers/counters: Timer 0 and Timer 1. They can be configured in various operating modes via the TMOD register. Logic diagram blocks internally convert oscillator pulses into time delays.

Methodology:

Calculating Timer Reload Values for Time Delays To generate a specific time delay using a timer in Mode 1 (16-bit timer mode), compute the initial value to load into THx and TLx.

1. Determine the machine cycle frequency (f_{mc}) and duration (T_{mc}). For an standard 8051, a machine cycle consists of 12 oscillator periods.

$$f_{mc} = \frac{f_{osc}}{12}$$

$$T_{mc} = \frac{1}{f_{mc}}$$

2. Calculate the total number of machine cycles required (N) for the desired delay (T_d).

$$N = \frac{T_d}{T_{mc}}$$

3. Subtract the number of cycles N from the maximum count plus one (65536 for 16-bit mode).

$$\text{Initial Count} = 65536 - N$$

4. Convert the Initial Count to a 16-bit hexadecimal number. The higher byte is loaded into THx, and the lower byte into TLx.

Worked Example:

Calculating TH0 and TL0 for Delay Calculate the TH0 and TL0 values required to generate a time delay of 5 ms using Timer 0 in Mode 1. Assume an oscillator frequency of 11.0592 MHz.
Solution:

1. **Calculate Machine Cycle Time (T_{mc}):**

$$f_{osc} = 11.0592 \text{ MHz}$$

$$f_{mc} = \frac{11.0592 \text{ MHz}}{12} = 921.6 \text{ kHz}$$

$$T_{mc} = \frac{1}{921.6 \text{ kHz}} \approx 1.085 \ \mu\text{s}$$

2. **Calculate Total Cycles Required (N):**

$$T_d = 5 \text{ ms} = 5000 \ \mu\text{s}$$

$$N = \frac{5000}{1.085} \approx 4608 \text{ cycles}$$

3. **Calculate Initial Count:**

$$\text{Initial Count} = 65536 - 4608 = 60928$$

4. **Convert to Hexadecimal:**

$$60928_{10} = \text{EE00}_{16}$$

Therefore, TH0 = `0xEE` and TL0 = `0x00`.

3.5 Serial Communication and Baud Rate Calculation

The 8051 contains an integrated UART for serial communication supporting multiple modes. In Mode 1 (8-bit UART with variable baud rate), Timer 1 is typically used in Mode 2 (8-bit auto-reload) to set the baud rate.

Methodology:

Calculating TH1 for Serial Baud Rate To compute the TH1 auto-reload value for a specific serial baud rate using Timer 1 in Mode 2:

1. Determine the UART clock frequency depending on the SMOD bit in the PCON register.

$$\text{UART Frequency} = \frac{f_{osc}}{12 \times 32} \times (2^{\text{SMOD}})$$

2. Calculate the TH1 auto-reload value required to generate the desired Baud Rate.

$$\text{TH1} = 256 - \left(\frac{\text{UART Frequency}}{\text{Baud Rate}} \right)$$

3. Ensure the fractional part is zero. Convert the integer result to hexadecimal.

Worked Example:

Calculating TH1 for 9600 Baud Determine the required TH1 value for a baud rate of 9600 bps using an 8051 with an oscillator frequency of 11.0592 MHz. Assume SMOD = 0.
Solution:

1. **Calculate UART Frequency:**

$$f_{osc} = 11.0592 \text{ MHz} = 11059200 \text{ Hz}$$

$$\text{UART Frequency} = \frac{11059200}{12 \times 32} \times (2^0) = \frac{11059200}{384} \times 1 = 28800 \text{ Hz}$$

2. **Calculate TH1 Value:**

$$\text{TH1} = 256 - \left(\frac{28800}{9600}\right) = 256 - 3 = 253$$

3. **Convert to Hexadecimal:**

$$253_{10} = \text{FD}_{16}$$

Therefore, the Timer 1 reload register TH1 must be loaded with `0xFD`.

3.6 Interrupt Vector Address and Priority

The 8051 provides five standard interrupts: External 0 (INT0), Timer 0 (TF0), External 1 (INT1), Timer 1 (TF1), and Serial Port (RI/TI). Each has a fixed vector address and a priority level.

Methodology:

Resolving Interrupt Priorities When multiple interrupts occur simultaneously, the 8051 resolves them using a predefined sequence:

1. Check the Interrupt Priority (IP) register. Interrupts configured for High Priority (bit = 1) are serviced before those configured for Low Priority (bit = 0).

2. If two pending interrupts have the same priority level, a polling sequence resolves the tie:

Interrupt Source	Flag	Vector Address
External Interrupt 0	IE0	`0x0003`
Timer 0 Interrupt	TF0	`0x000B`
External Interrupt 1	IE1	`0x0013`
Timer 1 Interrupt	TF1	`0x001B`
Serial Communication	RI/TI	`0x0023`

3. An interrupt in progress cannot be interrupted by another interrupt of the same or lower priority level.

Worked Example:

Determining Interrupt Execution Sequence Assume the IP register is configured such that Timer 1 and External 0 have high priority, while all others are low priority. Suppose Timer 0, External 0, and Serial interrupts all trigger at exactly the same time. Determine the order in which they will be serviced and state the corresponding vector addresses.
Solution:

1. **Identify Priority Levels:**

   - External 0 (INT0): High Priority
   - Timer 0 (TF0): Low Priority
   - Serial (RI/TI): Low Priority

2. **Select Highest Priority:** External 0 is the only active high-priority interrupt among the three. Therefore, it is serviced first.

3. **Resolve Ties within Priority Levels:** Once External 0 finishes, the microcontroller evaluates pending low-priority interrupts (Timer 0 and Serial). According to the default polling sequence, Timer 0 has a higher natural precedence than Serial.

4. **Determine Execution Sequence and Vector Addresses:**

   (a) **First:** External 0. Vector Address = `0x0003`

   (b) **Second:** Timer 0. Vector Address = `0x000B`

   (c) **Third:** Serial. Vector Address = `0x0023`

CHAPTER 4

8051 Programming

4.1 Addressing Modes

Methodology:

Identification of Addressing Modes To determine the addressing mode of an 8051 instruction mathematically and syntactically, analyze the operand systematically:

1. **Immediate Addressing:** Look for the hash symbol (#) preceding the operand. The data is provided directly as a constant (e.g., `MOV A, #45H`).
2. **Register Addressing:** Look for specific architectural registers like `A`, `B`, `DPTR`, or `R0` through `R7` (e.g., `ADD A, R2`).
3. **Direct Addressing:** Look for an 8-bit memory address or Special Function Register (SFR) name without any prefix (e.g., `MOV A, 30H` or `MOV P1, A`).
4. **Indirect Addressing:** Look for the at symbol (@) preceding `R0` or `R1`. The register computationally holds the memory address of the target data (e.g., `MOV A, @R0`).
5. **Indexed Addressing:** Look for a base register (`DPTR` or `PC`) computationally added to an offset in the Accumulator (`A`). Used for lookup tables (e.g., `MOVC A, @A+DPTR`).
6. **Relative Addressing:** Used exclusively with branching instructions. The operand is an 8-bit signed offset added mathematically to the Program Counter (e.g., `SJMP 05H`).
7. **Bit Addressing:** The operand points to a single bit address or a bit within a bit-addressable SFR (e.g., `SETB P1.2` or `CLR 20H`).

Worked Example:

Identifying Instruction Addressing Modes Identify the computational addressing mode for each of the following 8051 instructions:

1. `MOV R4, #55H`
2. `MOV A, 40H`
3. `MOVX A, @DPTR`
4. `MOVC A, @A+PC`
5. `SETB P2.0`

Solution:

1. `MOV R4, #55H`: The hash symbol (#) indicates the numeric data `55H` is provided directly. This is **Immediate Addressing**.

2. `MOV A, 40H`: The operand `40H` is a raw numerical address without a hash or '@' prefix. This is **Direct Addressing**.
3. `MOVX A, @DPTR`: The '@' symbol indicates the address is held in the DPTR register acting as a pointer. This is **Indirect Addressing**.
4. `MOVC A, @A+PC`: The address is calculated dynamically by adding the Accumulator and the Program Counter. This is **Indexed Addressing**.
5. `SETB P2.0`: The operation specifically targets a single computational bit (bit 0 of Port 2). This is **Bit Addressing**.

4.2 Instruction Set Categories

4.2.1 Data Transfer Instructions

Methodology:

Block Data Transfer Algorithms To move contiguous blocks of data in memory computationally, indirect addressing must be paired with loop structures.

1. Initialize a source pointer register (e.g., `R0`) to the starting address of the source block.
2. Initialize a destination pointer register (e.g., `R1`) to the starting address of the target block.
3. Initialize a loop counter register with the total number of bytes to transfer.
4. Inside the loop, read data from the source pointer into the Accumulator.
5. Write the Accumulator data to the destination pointer.
6. Increment both pointer registers mathematically.
7. Decrement the loop counter and repeat until the counter reaches zero.

Worked Example:

Block Data Transfer Loop Transfer 5 bytes of data stored in internal RAM starting at address `40H` to RAM locations starting at `50H`.
Solution: We will map registers `R0` and `R1` as memory pointers and `R2` as a decrementing loop counter.

```
        MOV R0, #40H    ; Source pointer
        MOV R1, #50H    ; Destination pointer
        MOV R2, #05H    ; Loop counter = 5
LOOP:   MOV A, @R0      ; Read byte from source
        MOV @R1, A      ; Write byte to destination
        INC R0          ; Increment source pointer
        INC R1          ; Increment destination pointer
        DJNZ R2, LOOP   ; Decrement counter and jump if not zero
```

Trace Analysis:

- **Iteration 1:** `R0=40H`, `R1=50H`. Data at `40H` moves to `50H`. `R0` becomes `41H`, `R1` becomes `51H`. `R2` becomes `04H` (Loop continues).
- Process dynamically repeats until `R2` reaches `00H`. The loop mathematically terminates after exactly 5 memory operations.

4.2.2 Arithmetic Instructions

Methodology:

Multibyte Addition Algorithm Since the 8051 ALU is strictly 8-bit, adding numbers larger than 8 bits computationally requires cascading through the Carry Flag (CY).

1. Load the lowest 8-bit byte of the first number into the Accumulator (A).
2. Add the lowest byte of the second number using the standard ADD instruction.
3. Store the resulting byte in memory. This operation may mathematically generate a carry out (CY=1).
4. Load the next higher byte of the first number into A.
5. Add the next higher byte of the second number using the ADDC (Add with Carry) instruction to computationally include any carry from the previous byte.
6. Store the resulting byte. Repeat for all subsequent bytes.

Worked Example:

Sixteen Bit Addition Write an assembly sequence to mathematically add two 16-bit numbers: 3A75H and 419AH. Store the result in R1 (upper byte) and R0 (lower byte).
Solution: Let Number 1 be 3A75H (Upper = 3AH, Lower = 75H).
Let Number 2 be 419AH (Upper = 41H, Lower = 9AH).
Mathematical Execution Steps:

1. Add lower bytes: 75H + 9AH = 10FH. The 8-bit sum is 0FH and the Carry Flag (CY) mathematically flips to 1.
2. Add upper bytes with carry: 3AH + 41H + 1 (from CY) = 7CH.

Assembly Implementation:

```
CLR C          ; Clear carry flag before computational addition
MOV A, #75H    ; Load lower byte of Num1
ADD A, #9AH    ; Add lower byte of Num2
MOV R0, A      ; Store lower byte of sum (0FH) in R0
MOV A, #3AH    ; Load upper byte of Num1
ADDC A, #41H   ; Add upper byte of Num2 with Carry flag
MOV R1, A      ; Store upper byte of sum (7CH) in R1
```

Final Result: R1=7CH, R0=0FH (Aggregate Sum = 7C0FH).

4.2.3 Logical Instructions Data Manipulation and Masking

Methodology:

Logical Bit Masking Algorithm Computational masking is deployed to isolate, clear, set, or toggle specific bits within an 8-bit data register without destructively affecting adjacent bits.

1. **Clearing Bits (AND Mask):** Apply the ANL A, #mask instruction. Set mask bits to 0 for targets you want to forcefully clear, and 1 for positions you want to preserve.
2. **Setting Bits (OR Mask):** Apply the ORL A, #mask instruction. Set mask bits to 1 for targets you want to forcefully set, and 0 for positions you want to preserve.
3. **Toggling Bits (XOR Mask):** Apply the XRL A, #mask instruction. Set mask bits to 1 for targets you want to mathematically invert, and 0 for positions you want to preserve.

Worked Example:

Masking the Lower Nibble Write a computational sequence to isolate the upper nibble of the Accumulator (clear the lower nibble), set the highest bit (bit 7) to 1, and toggle bit 4. Assume the initial state of A is B6H (1011 0110 in binary).
Solution: Step 1: Clear the lower nibble To mathematically clear bits 3-0 and preserve bits 7-4, the AND mask must be 1111 0000 (F0H).

```
ANL A, #0F0H  ; A = B6H AND F0H = B0H (1011 0000)
```

Step 2: Set bit 7 To strictly set bit 7 and leave others intact, the OR mask must be 1000 0000 (80H).

```
ORL A, #80H  ; A = B0H OR 80H = B0H (1011 0000) (already set)
```

Step 3: Toggle bit 4 To computationally invert bit 4, the XOR mask must be 0001 0000 (10H).

```
XRL A, #10H  ; A = B0H XOR 10H = A0H (1010 0000)
```

Final Accumulator State: A0H.

4.2.4 Branching and Conditional Programming

Methodology:

Conditional Array Parsing To computationally scan memory for a specific threshold or maximum using the CJNE (Compare and Jump if Not Equal) instruction:

1. Initialize a pointer to the base memory address and configure a loop counter.
2. Load the base element into the Accumulator as the "computational maximum".
3. Decrement the counter. If mathematically zero, terminate the algorithm.
4. Increment the pointer and parse the next memory byte into a secondary register (e.g., B).
5. Execute CJNE A, B, NEXT to trigger an arithmetic comparison.
6. If the Carry flag (CY) mathematically evaluates to 1, it dictates A $<$ B. Transfer B into A to update the maximum. If CY is 0, A $\geq$ B, so retain A.
7. Loop back to Step 3.

Worked Example:

Find the Largest Number Assume 3 numeric data points are stored in RAM addresses 30H, 31H, and 32H. Implement an algorithm to find the largest number and save it in 40H.
Solution:

```
        MOV R0, #30H    ; Base array pointer
        MOV R2, #03H    ; Element counter
        MOV A, #00H     ; Initialize global maximum to 0
AGAIN:  MOV B, @R0      ; Load subsequent element
        CJNE A, B, CHK  ; Mathematically compare A and B
CHK:    JNC NEXT        ; Jump if No Carry (A >= B)
        MOV A, B        ; Overwrite max value (A < B)
NEXT:   INC R0          ; Shift pointer to next address
        DJNZ R2, AGAIN  ; Decrement counter and conditionally loop
        MOV 40H, A      ; Write final maximum to 40H
```

Computational Trace: If RAM[30H]=10H, RAM[31H]=25H, RAM[32H]=1AH.

- **Iteration 1:** A=00H, B=10H. CJNE algorithmically sets CY=1 (00H < 10H). JNC falls through. A updates to 10H.
- **Iteration 2:** A=10H, B=25H. CJNE algorithmically sets CY=1. JNC falls through. A updates to 25H.
- **Iteration 3:** A=25H, B=1AH. CJNE algorithmically sets CY=0 (25H > 1AH). JNC executes jump. A statically remains 25H.

4.2.5 Machine Control Instructions

Methodology:

Machine Control and Time Delay Calculations Machine control instructions like NOP (No Operation) are utilized mathematically to consume machine cycles without altering any architectural registers. This is fundamental for building precise computational time delays.

1. **Machine Cycle Formula:** For an 8051 running at oscillator frequency f_{osc}, the time dimension for one machine cycle (T_{MC}) is calculated as:
$$T_{MC} = \frac{12}{f_{osc}}$$
2. **Delay Calculation:** An algorithmic block's execution time is its net cycle count multiplied by T_{MC}.
3. **NOP Insertion:** A NOP consumes exactly 1 machine cycle. Insert it into loops to fine-tune algebraic delays.

Worked Example:

Calculating Execution Time Calculate the total computational execution time for the given delay algorithm. Assume an 11.0592 MHz crystal oscillator driving the hardware.

```
        MOV R3, #200    ; 1 Machine Cycle
LOOP:   NOP             ; 1 Machine Cycle
        NOP             ; 1 Machine Cycle
        DJNZ R3, LOOP   ; 2 Machine Cycles
```

Solution: Step 1: Calculate Machine Cycle Time (T_{MC})

$$T_{MC} = \frac{12}{11.0592 \times 10^6} \approx 1.085\mu s$$

Step 2: Aggregate Total Machine Cycles for the Loop

- MOV R3, #200 executes strictly once: 1 total cycle.
- The looping body (NOP, NOP, DJNZ) consumes $1 + 1 + 2 = 4$ cycles per algorithmic iteration.
- Since R3 = 200, the body reiterates 200 times. Iteration cycles = $200 \times 4 = 800$ cycles.

Step 3: Calculate Total Time Dimension Total Machine Cycles = 1(init) + 800(loop iterations) = 801 cycles.
Total Execution Time = $801 \times 1.085\mu s = 869.085\mu s$.

4.3 Stack Operations

Methodology:

Tracking Stack Pointer and Memory The Stack Pointer (SP) mathematically tracks the active peak of the stack within internal RAM. Post-reset, SP auto-initializes to 07H.

1. **PUSH Operation Algorithm:** When `PUSH addr` is triggered:
   (a) SP is mathematically incremented by 1: SP $\leftarrow$ SP + 1.
   (b) The contents of the specified direct address are replicated into the internal RAM location designated by the newly calculated SP.
2. **POP Operation Algorithm:** When `POP addr` is triggered:
   (a) The contents of the internal RAM location dynamically pointed to by SP are routed to the specified direct address.
   (b) SP is mathematically decremented by 1: SP $\leftarrow$ SP $-$ 1.

Worked Example:

Executing Stack Instructions Assume an initial computational state of SP = 07H, R0 = 25H, and R1 = 3AH. Trace the dynamic values of SP and the stack memory state after systematically executing the following commands:

```
PUSH 00H   ; Push Address of R0 onto stack
PUSH 01H   ; Push Address of R1 onto stack
POP 02H    ; Pop top of stack into Address of R2
```

Solution: Step 1: Execute `PUSH 00H`

- SP algebraically increments: SP $= 07H + 1 = 08H$.
- Data from R0 (00H) is mapped to RAM address 08H. Memory block at 08H dynamically updates to 25H.

Step 2: Execute `PUSH 01H`

- SP algebraically increments: SP $= 08H + 1 = 09H$.
- Data from R1 (01H) is mapped to RAM address 09H. Memory block at 09H dynamically updates to 3AH.

Step 3: Execute `POP 02H`

- Data at RAM address 09H (3AH) is unspooled into R2 (02H). Now R2 structurally holds 3AH.
- SP algebraically decrements: SP $= 09H - 1 = 08H$.

Final Architectural State: SP = 08H, RAM[08H] = 25H, RAM[09H] = 3AH, R2 = 3AH.

CHAPTER 5

Interfacing and Applications of Microcontroller

Microcontrollers must interact with external physical devices to be useful in embedded systems. This chapter details the computational algorithms, mathematical formulations, and step-by-step methods required to interface external peripherals with the 8051 microcontroller.

5.1 LED and Push Button Switch Interfacing

Interfacing basic Input/Output (I/O) components requires configuring the port pins and using polling mechanisms to read inputs and drive outputs.

Methodology:

Push Button and LED Logic Algorithm To correctly read a switch and actuate an LED, follow these steps:

1. **Configure Input**: Write a logic `1` to the specific port pin where the switch is connected. This configures the pin as an input.
2. **Debounce and Read**: Read the status of the pin. If the switch pulls the pin to ground when pressed, a logic `0` indicates activation.
3. **Drive Output**: Write a logic `1` (for current sourcing) or `0` (for current sinking) to the port pin connected to the LED.

Worked Example:

Switch Controlled LED **Problem:** An LED is connected to pin P1.0 and a push button switch is connected to pin P2.0 of an 8051 microcontroller. Write an algorithm to turn ON the LED when the switch is pressed and turn it OFF when released.
Solution:

1. **Initialization:** Set P2.0 as input.
   `SETB P2.0` (Write 1 to P2.0)
2. **Monitor Switch:** Check the status of P2.0.
   `JNB P2.0, TURN_ON` (If P2.0 is 0, switch is pressed, jump to TURN_ON)
3. **Turn OFF LED:** If not pressed, clear P1.0.
   `CLR P1.0` (LED OFF)
   `SJMP MONITOR` (Repeat monitoring)
4. **Turn ON LED (TURN_ON routine):**
   `SETB P1.0` (LED ON)
   `SJMP MONITOR` (Repeat monitoring)

5.2 Relay Interfacing

A relay acts as an electromechanical switch to control high voltage/current devices. Since the microcontroller cannot provide sufficient driving current, a driver transistor (or ULN2003 IC) is used.

Methodology:

Relay Actuation Method The control logic for a relay is mathematically simple but requires current amplification:

1. Output a `HIGH (1)` from the microcontroller pin to the base of an NPN driver transistor to turn the transistor ON.
2. The transistor acts as a closed switch pulling the relay coil to ground, energizing it.
3. Output a `LOW (0)` to turn the transistor OFF, de-energizing the relay.

Worked Example:

Time Delayed Relay Control **Problem:** Write a process to turn ON a relay connected to P1.5 for a specific duration and then turn it OFF.
Solution:

1. **Turn Relay ON:** Send high signal to driver.
   `SETB P1.5`
2. **Wait:** Call a delay subroutine representing the required ON-time duration.
   `ACALL DELAY`
3. **Turn Relay OFF:** Send low signal to driver.
   `CLR P1.5`

5.3 7-Segment Display Interfacing

A 7-segment display requires specific hexadecimal codes to turn on the appropriate segments (A through G) to display decimal digits.

Methodology:

Lookup Table Generation for 7-Segment To display a digit $D \in \{0, 1, \ldots, 9\}$, a lookup table is used to map the digit to its segment code.

1. Define segment mapping (assuming Common Cathode, where `1` turns a segment ON and `0` turns it OFF):

Digit	dp	g	f	e	d	c	b	a
0	0	0	1	1	1	1	1	1 (Hex: 3F)
1	0	0	0	0	0	1	1	0 (Hex: 06)

2. Store these Hex values sequentially in an array or ROM memory block.
3. Fetch the N-th entry of the table to display digit N.

Worked Example:

Single Digit Counter Algorithm **Problem:** Display digits 0 to 9 continuously on a common cathode 7-segment display connected to Port 2.

Solution:

1. **Define Table:** Load the base address of the lookup table.
   `MOV DPTR, #LUT`
2. **Initialize Counter:** Set a register (e.g., R0) to 0.
3. **Fetch Code:** Move counter value to Accumulator.
   `MOV A, R0`
   `MOVC A, @A+DPTR`
4. **Display Code:** Send Accumulator data to Port 2.
   `MOV P2, A`
5. **Delay and Increment:** Call delay and increment R0. If R0 reaches 10 reset it to 0. Repeat.

5.4 Liquid Crystal Display Interfacing

A 16x2 LCD module requires initialization commands followed by data writing to display characters. The communication uses RS (Register Select), RW (Read/Write), and EN (Enable) control lines.

Methodology:

LCD Command and Data Writing Steps **Command Write Cycle (RS=0):**

1. Place the command byte on the data port (e.g., P1).
2. Clear RS pin (`RS = 0`) to select the Instruction Register.
3. Clear RW pin (`RW = 0`) to write operation.
4. Apply a High-to-Low pulse on EN pin (`EN = 1`, brief delay, `EN = 0`).

Data Write Cycle (RS=1):

1. Place the character ASCII byte on the data port.
2. Set RS pin (`RS = 1`) to select the Data Register.
3. Clear RW pin (`RW = 0`) to write operation.
4. Apply a High-to-Low pulse on EN pin.

Worked Example:

Display Character A on LCD **Problem:** Output the character 'A' (ASCII 41H) on an LCD. Data port is P1 RS is P2.0 RW is P2.1 EN is P2.2.
Solution:

1. **Send Initialization Commands:** Write 38H (2 lines 5x7 matrix) Write 0EH (Display ON Cursor ON) Write 01H (Clear display) using the Command Write Cycle.
2. **Write Character 'A':** `MOV P1, #41H` (Load ASCII for 'A') `SETB P2.0` (RS=1 for Data) `CLR P2.1` (RW=0 for Write) `SETB P2.2` (EN=1) `ACALL DELAY CLR P2.2` (EN=0 pulse completes write)

5.5 DAC0808 Interfacing

A Digital-to-Analog Converter (DAC) like the DAC0808 takes an 8-bit digital input and produces an equivalent analog output current, which is converted to voltage using an op-amp.

Methodology:

DAC Voltage Calculation and Waveform Generation The output voltage V_{out} for an 8-bit DAC is given by:

$$V_{out} = V_{ref} \times \left(\frac{D}{256}\right)$$

where V_{ref} is the reference voltage and D is the decimal equivalent of the 8-bit digital input (0 to 255).
Waveform Generation Algorithm:

1. Output a sequence of values D to the port connected to the DAC.
2. To generate a **Square wave**: Output 00H, wait, Output FFH, wait, repeat.
3. To generate a **Triangular wave**: Increment D from 00H to FFH linearly, then decrement D from FFH down to 00H linearly.

Worked Example:

Compute DAC Output Voltage **Problem:** A DAC0808 is interfaced with an 8051. If $V_{ref} = 5$V, calculate the output voltage when the digital input is 80H.
Solution:

1. Convert the hexadecimal input to decimal:
   $80\text{H} = (8 \times 16^1) + (0 \times 16^0) = 128_{10}$
2. Apply the DAC formula:
   $V_{out} = 5 \times \left(\frac{128}{256}\right)$
3. Compute the final voltage:
   $V_{out} = 5 \times 0.5 = 2.5\text{V}$

5.6 ADC0804 Interfacing

The Analog-to-Digital Converter (ADC0804) converts analog voltage into an 8-bit digital value. The conversion time depends on the clock frequency.

Methodology:

ADC Data Conversion Process The protocol to read digital data from ADC0804 involves the CS (Chip Select), WR (Write), INTR (Interrupt), and RD (Read) pins:

1. **Start Conversion:** Send a LOW-to-HIGH pulse to the WR pin while holding CS LOW.
2. **Wait for End of Conversion:** Poll the INTR pin. When it transitions to LOW (`0`), the conversion is complete.
3. **Read Data:** Pull CS and RD pins LOW to output the digital 8-bit data onto the data pins, and read the port into the microcontroller.

Worked Example:

Reading Digital Value from ADC **Problem:** Write the operational steps to fetch one byte of converted data from ADC0804. Assume WR=P2.6 INTR=P2.5 RD=P2.7 and Data=Port 1.
Solution:

1. **Start Pulse:**
   `CLR P2.6` (WR=0)
   `SETB P2.6` (WR=1 start conversion)
2. **Wait for INTR:**
   `JB P2.5, $` (Wait here while INTR is 1)
3. **Read Output:**
   `CLR P2.7` (RD=0 to enable output)
   `MOV A, P1` (Read Port 1 into Accumulator)
   `SETB P2.7` (RD=1 to finish read)

5.7 DC Motor Interfacing

A DC motor requires high current and must often run bidirectionally. A motor driver IC like the L293D is used. Speed control is achieved using Pulse Width Modulation (PWM).

Methodology:

DC Motor Direction Control Logic An H-bridge motor driver has two input pins (IN1, IN2) for each motor. The motor response depends on the binary states applied to these pins.

IN1	IN2	Motor Operation
0	0	Stop
1	0	Clockwise Rotation
0	1	Anti-Clockwise Rotation
1	1	Fast Motor Stop

Worked Example:

Rotate DC Motor with L293D **Problem:** Write the port operations to rotate a DC motor clockwise for a fixed duration and then stop it. Assume IN1 = P1.0 and IN2 = P1.1.
Solution:

1. **Rotate Clockwise:**
   `SETB P1.0` (IN1 = 1)
   `CLR P1.1` (IN2 = 0)
2. **Maintain State:** Call a delay to keep the motor running.
3. **Stop Motor:**
   `CLR P1.0` (IN1 = 0)
   `CLR P1.1` (IN2 = 0)

5.8 Stepper Motor Interfacing

A stepper motor rotates in precise discrete steps. Interfacing involves energizing the motor's coils in a specific sequence using a driver like ULN2003.

Methodology:

Step Sequence Generation Formula The total number of steps required to reach a target angle θ_{target} is calculated as:

$$\text{Step Angle} = \frac{360^\circ}{\text{Number of Rotor Teeth}}$$

$$\text{Number of Steps} = \frac{\theta_{target}}{\text{Step Angle}}$$

To rotate the motor, a standard 4-step (full-step) sequence must be cyclically applied to the 4 coil inputs:

Step	Coil 4	Coil 3	Coil 2	Coil 1	Hex Code
1	1	0	0	1	09H
2	1	1	0	0	0CH
3	0	1	1	0	06H
4	0	0	1	1	03H

Worked Example:

Stepper Motor Rotation Calculation **Problem:** A stepper motor has a step angle of 1.8°. Calculate the number of steps required to rotate the motor by 90°. Write the logic to perform 1 complete step cycle.
Solution:

1. **Calculate Steps:**
   Number of Steps = $\frac{90^\circ}{1.8^\circ}$ = 50 steps.
2. **Output Sequence (to Port 1):**
   `MOV P1, #09H` Wait 10ms
   `MOV P1, #0CH` Wait 10ms
   `MOV P1, #06H` Wait 10ms
   `MOV P1, #03H` Wait 10ms
3. **Repeat** this 4-step sequence 12 times and perform 2 additional steps to reach the total 50 steps.

5.9 Applications of Microcontrollers in Various Fields

Microcontrollers solve complex problems in various fields by combining the interfacing techniques discussed above.

Methodology:

Embedded System Application Design Cycle To solve an application problem computationally using a microcontroller:

1. **Identify Inputs:** Select appropriate sensors (Temperature, Light, Push Buttons) and pass them through ADCs or digital input ports.
2. **Define Computation logic:** Implement thresholds, timers, and PID loops in the microcontroller firmware.
3. **Actuate Outputs:** Use DACs, Relays, and Motor Drivers to trigger physical responses (e.g., cooling fans, alarms).
4. **Provide Feedback:** Display system states on LCDs or 7-Segment Displays.

Worked Example:

Design of a Temperature Control System **Problem:** Formulate the step-by-step logic for a room temperature monitor that turns on a DC cooling fan when the temperature exceeds 30°C.
Solution:

1. **Input Stage:** Interfaced LM35 temperature sensor provides analog voltage.
2. **Conversion Phase:** ADC0804 converts the analog voltage into an 8-bit digital value.
3. **Computation Phase:** The microcontroller reads the ADC data. It compares the data against a stored digital threshold corresponding to 30°C.
4. **Output Stage:** If the value is > threshold, the microcontroller sends a logic `1` to the L293D motor driver, spinning the DC fan.
5. **Display:** The current temperature reading is mapped via a lookup table and displayed on a 16x2 LCD.

Formulae

Appendix: Key Formulae

This appendix provides a quick reference to the key theoretical and mathematical formulae used for problem-solving across all units of this book.

Unit 1: Microprocessor Architecture & Memory Interfacing

Clock and Timing

The timing characteristics of a microprocessor depend heavily on its oscillator frequency.

- **Clock Period (T):**
$$T = \frac{1}{f}$$
where f is the operating clock frequency.
- **Instruction Execution Time (T_{exec}):**
$$T_{exec} = N_T \times T = \frac{N_T}{f}$$
where N_T is the total number of T-states required to fetch and execute the instruction.

Memory and Interfacing

Calculations to determine total addressable memory, capacity, and the chips required for building a target memory system.

- **Total Memory Locations:**
$$\text{Locations} = 2^N$$
where N is the number of address lines available.
- **Memory Capacity:**
$$\text{Capacity} = \text{Total Locations} \times \text{Word Size}$$
- **Total Number of Memory Chips Required (N_{chips}):**
$$N_{chips} = \frac{\text{Target Memory Capacity}}{\text{Capacity of a Single Chip}}$$
- **Memory Array Configuration:**
$$\text{Number of Rows} = \frac{\text{Target Number of Words}}{\text{Words per Chip}}$$
$$\text{Number of Columns (Parallel Chips)} = \frac{\text{Target Word Size}}{\text{Chip Word Size}}$$
- **Address Mapping:**
$$\Delta = \text{Total Locations (Hex)} - 1\text{H}$$
$$A_{end} = A_{start} + \Delta$$
where A_{start} is the starting address and A_{end} is the ending address of the memory block.

Unit 2: 8051 Microcontroller Architecture

Oscillator and Machine Cycle

- **Machine Cycle Frequency (f_{mc}):**

$$f_{mc} = \frac{f_{osc}}{12}$$

  where f_{osc} is the crystal oscillator frequency.

- **Machine Cycle Time (T_{mc}):**

$$T_{mc} = \frac{1}{f_{mc}} = \frac{12}{f_{osc}}$$

Memory Addressing Calculations

- **Register Bank Base Address:**

$$\text{Base Address} = \text{Bank Number} \times 8$$

- **Absolute Address of a Register:**

$$\text{Absolute Address} = \text{Base Address} + R_n$$

  where R_n is the register number (0 to 7).

- **Bit-Addressable RAM Calculation:**

$$\text{Byte Offset} = \text{Byte Address} - 20\text{H}$$

$$\text{Bit Address (Decimal)} = (\text{Byte Offset}_{10} \times 8) + \text{Bit Number}$$

Unit 3: 8051 Instruction Set

Stack Operations

The stack pointer (SP) in the 8051 points to the last used location.

- **PUSH Operation:**

$$\text{SP} \leftarrow \text{SP} + 1$$

  Data is then written to the internal RAM location addressed by the new SP.

- **POP Operation:** Data is read from the internal RAM location addressed by the current SP.

$$\text{SP} \leftarrow \text{SP} - 1$$

Unit 4: Timers, Serial Port, and Interrupts

Timer Delay Calculations

- **Timer Count (N):**

$$N = \frac{\text{Required Delay}}{T_{mc}}$$

- **Timer Reload Value:**

$$\text{Reload Value (8-bit mode)} = 256 - N$$

$$\text{Reload Value (16-bit mode)} = 65536 - N$$

Serial Communication

- **Baud Rate Calculation (Mode 1):**

$$\text{Baud Rate} = \frac{2^{\text{SMOD}}}{32} \times \frac{f_{osc}}{12 \times (256 - \text{TH1})}$$

- **Timer 1 Reload Value (TH1) for Target Baud Rate:**

$$\text{TH1} = 256 - \left(\frac{2^{\text{SMOD}} \times f_{osc}}{384 \times \text{Baud Rate}} \right)$$

Special Function Registers (SFRs) Formats

- **TMOD (Timer/Counter Mode Control Register):**

$$\text{TMOD} = [\text{GATE1}]\ [\text{C/T1}]\ [\text{M1}_1]\ [\text{M0}_1] \quad [\text{GATE0}]\ [\text{C/T0}]\ [\text{M1}_0]\ [\text{M0}_0]$$

- **TCON (Timer/Counter Control Register):**

$$\text{TCON} = [\text{TF1}]\ [\text{TR1}]\ [\text{TF0}]\ [\text{TR0}] \quad [\text{IE1}]\ [\text{IT1}]\ [\text{IE0}]\ [\text{IT0}]$$

- **IE (Interrupt Enable Register):**

$$\text{IE} = [\text{EA}]\ [-]\ [\text{ET2}]\ [\text{ES}] \quad [\text{ET1}]\ [\text{EX1}]\ [\text{ET0}]\ [\text{EX0}]$$

- **IP (Interrupt Priority Register):**

$$\text{IP} = [-]\ [-]\ [\text{PT2}]\ [\text{PS}] \quad [\text{PT1}]\ [\text{PX1}]\ [\text{PT0}]\ [\text{PX0}]$$

Unit 5: 8051 Interfacing

Digital to Analog Converter (DAC)

- **DAC Output Voltage (V_{out}):**

$$V_{out} = V_{ref} \times \left(\frac{D}{2^n} \right)$$

where V_{ref} is the reference voltage, D is the digital input value (decimal), and n is the resolution in bits (e.g., $n = 8$ gives $2^8 = 256$ steps).

Stepper Motor Interfacing

- **Step Angle:**

$$\text{Step Angle} = \frac{360^\circ}{\text{Number of Rotor Teeth}}$$

- **Steps per Revolution:**

$$\text{Total Steps} = \frac{360^\circ}{\text{Step Angle}}$$

- **Target Step Calculation:**

$$\text{Number of Steps Required} = \frac{\theta_{target}}{\text{Step Angle}}$$

where θ_{target} is the target rotation angle in degrees.

- **Stepper Motor Full-Step Sequence:** Standard 4-step excitation sequence for coils D, C, B, A:

$$\begin{aligned}
&\text{Step 1:} \quad 03\text{H} \quad (0000\,0011_2) \\
&\text{Step 2:} \quad 06\text{H} \quad (0000\,0110_2) \\
&\text{Step 3:} \quad 0C\text{H} \quad (0000\,1100_2) \\
&\text{Step 4:} \quad 09\text{H} \quad (0000\,1001_2)
\end{aligned}$$