Mpmc (1333202) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Unit 1: Microprocessor Based Systems

1.1 Introduction

This chapter covers the fundamental mathematical calculations required in microprocessor and microcontroller-based systems. While the introductory concepts often focus on architecture, practical problem-solving involves calculating memory sizing, address bus width requirements, memory mapping ranges, and execution time analysis.

1.2 Memory Capacity and Address Lines

Methodology:

Calculating Memory Capacity The address bus width of a microprocessor determines the maximum number of unique memory locations it can access.

1. Identify the number of address lines (N) available in the microprocessor or microcontroller.
2. Calculate the total number of memory locations using the exponential formula:

$$\text{Total Locations} = 2^N$$

3. Convert the raw result into standard memory units (KB, MB, GB) by factoring out powers of 2:
   - 1 KB (Kilobyte) = 2^{10} bytes = 1024 bytes
   - 1 MB (Megabyte) = 2^{20} bytes = 1024 KB
   - 1 GB (Gigabyte) = 2^{30} bytes = 1024 MB
4. If the processor has an 8-bit (1 byte) data bus, the memory capacity is exactly 2^N bytes.

Worked Example:

Memory Capacity for 16-bit Address Bus A typical 8-bit microprocessor has a 16-bit address bus. Calculate the maximum memory capacity it can access in Kilobytes (KB).

Step 1: Identify given values.

- Number of address lines (N) = 16
- Data bus width = 8 bits = 1 byte (meaning each location holds 1 byte)

Step 2: Calculate total memory locations.

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

Step 3: Convert to standard memory units. We can split the exponent 2^{16} into $2^6 \times 2^{10}$. Since 2^{10} bytes = 1 KB and $2^6 = 64$:

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

Final Answer: The microprocessor can access up to 64 KB of memory.

Worked Example:

Determining Required Address Lines A microcontroller requires 256 KB of embedded flash memory. How many address lines are needed to fully map this memory?

Step 1: Convert the memory capacity into powers of 2.

$$256 \text{ KB} = 256 \times 1024 \text{ bytes}$$

Since $256 = 2^8$ and $1024 = 2^{10}$:

$$256 \text{ KB} = 2^8 \times 2^{10} = 2^{18} \text{ bytes}$$

Step 2: Equate to the formula for address lines.

$$2^N = 2^{18}$$

Step 3: Solve for the exponent N.

$$N = 18$$

Final Answer: 18 address lines are required to map 256 KB of memory.

1.3 Memory Mapping and Address Ranges

Methodology:

Calculating Ending Address of a Memory Block In microprocessor systems memory is often divided into blocks (like RAM and ROM). Given a starting hexadecimal address and the block size in bytes the ending address can be calculated mathematically.

1. Convert the given memory block size into its total number of bytes in decimal.
2. Convert the decimal byte size into its hexadecimal equivalent.
3. Use the mapping formula to find the ending address:

$$\text{Ending Address} = \text{Starting Address} + \text{Hexadecimal Size} - 1$$

4. Perform the addition and subtraction strictly using hexadecimal arithmetic.

Worked Example:

Finding the Ending Address A 4 KB RAM chip is interfaced to a microprocessor system. If its starting address is assigned as `2000H` determine its final ending address.

Step 1: Convert block size to bytes in decimal.

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

Step 2: Convert decimal size to hexadecimal. To convert 4096 to hexadecimal successively divide by 16:

Division	Quotient	Remainder (Hex)
$4096 \div 16$	256	0
$256 \div 16$	16	0
$16 \div 16$	1	0
$1 \div 16$	0	1

Reading the remainders from bottom to top $4096_{10} =$ `1000H`.

Step 3: Apply the ending address formula.

$$\text{Ending Address} = \text{Starting Address} + \text{Size} - 1$$

$$\text{Ending Address} = \texttt{2000H} + \texttt{1000H} - 1$$

$$\text{Ending Address} = \texttt{3000H} - 1$$

$$\text{Ending Address} = \texttt{2FFFH}$$

Final Answer: The ending address of the RAM block is `2FFFH`.

Worked Example:

Finding Total Memory Given an Address Range The memory address range for a specific EPROM is given as `8000H` to `BFFFH`. Calculate the total memory size of this EPROM in Kilobytes (KB).

Step 1: Determine the total number of bytes in hexadecimal.

$$\text{Total Bytes (Hex)} = \text{Ending Address} - \text{Starting Address} + 1$$

$$\text{Total Bytes (Hex)} = \texttt{BFFFH} - \texttt{8000H} + 1$$

Step 2: Perform the hexadecimal subtraction.

$$\texttt{BFFFH} - \texttt{8000H} = \texttt{3FFFH}$$

Add 1 to `3FFFH`:

$$\texttt{3FFFH} + 1 = \texttt{4000H}$$

Step 3: Convert the hexadecimal size to decimal. Multiply each hex digit by its positional power of 16:

$$\texttt{4000H} = (4 \times 16^3) + (0 \times 16^2) + (0 \times 16^1) + (0 \times 16^0)$$

$$\texttt{4000H} = 4 \times 4096 = 16384 \text{ bytes}$$

Step 4: Convert bytes to Kilobytes.

$$\frac{16384 \text{ bytes}}{1024 \text{ bytes/KB}} = 16 \text{ KB}$$

Final Answer: The total size of the EPROM is 16 KB.

1.4 Instruction Execution Time Analysis

Methodology:

Calculating Execution Time The physical time required to execute an instruction depends directly on the processor's clock frequency and the total number of internal clock cycles (T-states) dictated by the instruction's microcode.

1. Identify the clock frequency (f) of the microprocessor in Hertz (Hz).
2. Calculate the time duration of one single clock cycle (T-state) using the inverse formula:

$$T = \frac{1}{f}$$

3. Determine the total number of T-states (N_T) required to fetch decode and execute the instruction.
4. Calculate the total execution time (T_{exec}) by multiplying the T-states by the cycle duration:

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

Worked Example:

Execution Time of a Single Instruction A microprocessor operates at a master clock frequency of 3 MHz. A specific memory transfer instruction requires 13 T-states to execute completely. Calculate the exact execution time of the instruction in microseconds (μs).

Step 1: Identify given values.

- Clock frequency $f = 3$ MHz $= 3 \times 10^6$ Hz
- Number of T-states $N_T = 13$

Step 2: Calculate the duration of one T-state.

$$T = \frac{1}{f} = \frac{1}{3 \times 10^6} \text{ seconds}$$

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

Step 3: Calculate the total execution time.

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

$$T_{exec} = 13 \times \left(\frac{1}{3 \times 10^6}\right)$$

$$T_{exec} = \frac{13}{3} \mu s \approx 4.33 \mu s$$

Final Answer: The execution time of the instruction is approximately $4.33 \mu s$.

Worked Example:

Determining Required Clock Frequency A real-time control application requires a specific task involving a subroutine loop of 50 T-states to execute in exactly $25 \mu s$. What should be the minimum operating clock frequency of the microprocessor?

Step 1: Identify given values.

- Total execution time $T_{exec} = 25 \mu s = 25 \times 10^{-6}$ seconds
- Number of T-states $N_T = 50$

Step 2: Set up the execution time formula.

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

Substitute the known values:

$$25 \times 10^{-6} = 50 \times T$$

Step 3: Solve for the duration of one T-state (T).

$$T = \frac{25 \times 10^{-6}}{50}$$

$$T = 0.5 \times 10^{-6} \text{ seconds}$$

Step 4: Calculate the required clock frequency (f).

$$f = \frac{1}{T}$$

$$f = \frac{1}{0.5 \times 10^{-6}} = 2 \times 10^6 \text{ Hz}$$

$$f = 2 \text{ MHz}$$

Final Answer: The required clock frequency for the microprocessor is 2 MHz.

1.5 Microprocessor Performance

Methodology:

Calculating Instructions Per Second To evaluate raw computational performance it is useful to determine the average number of instructions a processor executes per second commonly expressed in MIPS (Millions of Instructions Per Second).

1. Determine the operating clock frequency (f) of the microprocessor in Hz.
2. Determine the average number of T-states (Clock Cycles Per Instruction or CPI) required per typical instruction.
3. Calculate the total number of instructions per second:

$$\text{Instructions Per Second} = \frac{f}{\text{CPI}}$$

4. Convert the result to MIPS by dividing by $1,000,000$ (10^6):

$$\text{MIPS} = \frac{\text{Instructions Per Second}}{10^6}$$

Worked Example:

Calculating MIPS for a Microcontroller An embedded microcontroller operates at a frequency of 16 MHz. Assuming an average instruction profile takes 4 clock cycles to fetch and execute calculate the processor's performance in MIPS.

Step 1: Identify given values.

- Clock frequency $f = 16$ MHz $= 16 \times 10^6$ Hz
- Average T-states per instruction (CPI) $= 4$

Step 2: Calculate total instructions per second.

$$\text{Instructions Per Second} = \frac{16 \times 10^6}{4}$$

$$\text{Instructions Per Second} = 4 \times 10^6$$

Step 3: Convert the raw value to MIPS.

$$\text{MIPS} = \frac{4 \times 10^6}{10^6} = 4 \text{ MIPS}$$

Final Answer: The microcontroller executes at a rate of 4 MIPS.

CHAPTER 2

Unit 2: 8085 Assembly Language Programming

Assembly language programming for the 8085 microprocessor involves using mnemonics to represent machine-level instructions. This chapter focuses on the computational methods and algorithmic techniques used to solve standard programming problems in 8085 assembly.

2.1 Time Delay Calculations

Microprocessors operate at a specific clock frequency. The time taken to execute an instruction is measured in T-states (Time states) where one T-state corresponds to one clock cycle. We can create precise time delays by writing loops that keep the processor busy for a calculated number of T-states.

Methodology:

Time Delay Calculation for a Single Loop To calculate the total time delay generated by a loop in 8085 assembly:

1. Identify the operating clock frequency f of the microprocessor.
2. Calculate the duration of one T-state: $T = \frac{1}{f}$.
3. Determine the total T-states for instructions outside the loop (Initialization T-states T_{init}).
4. Determine the total T-states for instructions inside the loop (T_{loop}).
5. Note the decimal count value N loaded into the loop counter register.
6. Calculate the total delay time T_{delay} using the formula:

$$T_{delay} = T \times (T_{init} + N \times T_{loop} - 3)$$

Note: The -3 accounts for the final iteration where the conditional jump is not taken (which takes 7 T-states instead of 10 T-states).

Worked Example:

Calculate the time delay for a specific 8085 program snippet Given the following 8085 program snippet and a clock frequency of 2 MHz calculate the total time delay.

Label	Instruction	T-States
	MVI C, 64H	7
LOOP:	DCR C	4
	JNZ LOOP	10/7

Step 1: Calculate the duration of one T-state.
Clock frequency $f = 2$ MHz $= 2 \times 10^6$ Hz.
Time period $T = \frac{1}{f} = \frac{1}{2\times10^6} = 0.5 \times 10^{-6}$ s $= 0.5\,\text{ţs}$.

Step 2: Identify Initialization and Loop T-states.
Initialization instruction: `MVI C, 64H` takes $T_{init} = 7$ T-states.
Loop instructions: `DCR C` (4 T-states) + `JNZ LOOP` (10 T-states) $\rightarrow T_{loop} = 14$ T-states.
Step 3: Identify the loop count N.
The counter is initialized with 64H. We must convert this to decimal.
$N = 64\text{H} = (6 \times 16) + 4 = 96 + 4 = 100_{10}$.
Step 4: Calculate the total delay time.

$$\begin{aligned} T_{delay} &= T \times (T_{init} + N \times T_{loop} - 3) \\ &= 0.5\,\mu\text{s} \times (7 + 100 \times 14 - 3) \\ &= 0.5\,\mu\text{s} \times (7 + 1400 - 3) \\ &= 0.5\,\mu\text{s} \times (1404) \\ &= 702\,\mu\text{s} \end{aligned}$$

The total time delay generated is 702 μs.

2.2 Arithmetic Operations

Arithmetic programs form the foundation of microprocessor-based computation. Since the 8085 is an 8-bit processor it operates natively on 8-bit data.

Methodology:

Eight Bit Addition with Carry To add two 8-bit numbers where the sum may exceed 8 bits (generating a carry):

1. Clear a register (e.g. Register C) to use as a carry counter (`MVI C, 00H`).
2. Load the first operand into the Accumulator from memory.
3. Move the second operand into a general-purpose register.
4. Add the register to the Accumulator (`ADD`).
5. Check the Carry flag. If no carry is generated jump over the next instruction (`JNC`).
6. If a carry is generated increment the carry register (`INR C`).
7. Store the Accumulator (lower byte of sum) in memory.
8. Store the carry register (upper byte of sum) in the next memory location.

Worked Example:

Add two 8-bit numbers 8AH and 9BH Write an 8085 assembly program to add 8AH and 9BH stored at 2000H and 2001H respectively. Store the sum at 2002H and the carry at 2003H. Trace the execution.
Program:

Address	Mnemonics	Comments
2100	MVI C, 00H	Initialize carry register C to 00H.
2102	LXI H, 2000H	Load HL pair with address 2000H.
2105	MOV A, M	Move first number (8AH) to Accumulator.
2106	INX H	Increment HL pair to point to 2001H.
2107	ADD M	Add second number (9BH) to Accumulator.
2108	JNC 210CH	If no carry jump to 210CH (Skip increment).
210B	INR C	Increment C if carry is generated.
210C	INX H	Increment HL pair to 2002H.
210D	MOV M, A	Store sum (Accumulator) at 2002H.
210E	INX H	Increment HL pair to 2003H.
210F	MOV M, C	Store carry (Register C) at 2003H.
2110	HLT	Halt the program.

Execution Trace:

1. `C = 00H`, `HL = 2000H`.
2. `A = [2000H] = 8AH`.
3. `HL = 2001H`.
4. `ADD M`: Add `A` and `[2001H]`. $8AH + 9BH = 10001010_2 + 10011011_2$. Sum $= 100100101_2$. Accumulator becomes 25H and Carry flag (CY) is set to 1.
5. `JNC 210CH`: Since CY=1 the jump is not taken.
6. `INR C`: Register `C` becomes `01H`.
7. `HL = 2002H`, store `A` (25H) at 2002H.
8. `HL = 2003H`, store `C` (01H) at 2003H.

Final result: Sum = 0125H. Address 2002H holds 25H, and 2003H holds 01H.

Methodology:

Sixteen Bit Arithmetic Addition To add two 16-bit numbers using an 8-bit microprocessor:

1. Store the 16-bit operands in memory (lower byte first followed by higher byte).
2. Load the lower byte of the first operand into the Accumulator.
3. Add the lower byte of the second operand to the Accumulator (`ADD`).
4. Store the resulting lower byte of the sum in memory.
5. Load the upper byte of the first operand into the Accumulator.
6. Add the upper byte of the second operand along with any carry from the lower byte addition (`ADC`).
7. Store the resulting upper byte of the sum in memory.

Worked Example:

Add two 16-bit numbers 1234H and 5678H Perform 16-bit addition of 1234H and 5678H. Assume 1234H is stored at 2000H-2001H and 5678H is stored at 2002H-2003H. Store result at 2004H-2005H.
Memory Layout Before Execution:

Address	Data
2000H	34H (Lower byte of 1st number)
2001H	12H (Upper byte of 1st number)
2002H	78H (Lower byte of 2nd number)
2003H	56H (Upper byte of 2nd number)

Execution Steps:

1. **Add Lower Bytes:**
   Load 34H into Accumulator. Add 78H.
   $34H + 78H = ACH$. No carry is generated (CY = 0).
   Store ACH at 2004H.

2. **Add Upper Bytes:**
   Load 12H into Accumulator. Add 56H with carry (which is 0).
   $12H + 56H + 0 = 68H$.
   Store 68H at 2005H.

Final Result in Memory:

Address	Data
2004H	ACH (Lower byte of result)
2005H	68H (Upper byte of result)

The final 16-bit sum is 68ACH.

Methodology:

Eight Bit Multiplication via Successive Addition The 8085 does not have a native multiply instruction. Multiplication of two 8-bit numbers can be performed by adding the multiplicand to itself a number of times equal to the multiplier.

1. Initialize an Accumulator to 00H to store the lower byte of the product.
2. Initialize another register (e.g. Register D) to 00H to store the upper byte of the product (carry tracking).
3. Load the multiplier into a counter register (e.g. Register C).
4. Check if the multiplier is zero. If yes skip the addition process.
5. Load the multiplicand into a temporary register (e.g. Register B).
6. Add the multiplicand (Register B) to the Accumulator.
7. If a carry is generated increment the upper byte register (Register D).
8. Decrement the counter (Register C).
9. If the counter is not zero jump back to step 6.
10. Store the Accumulator (lower byte) and Register D (upper byte) in memory.

Worked Example:

Multiply 05H and 03H Perform 8-bit multiplication of 05H (multiplicand) and 03H (multiplier).
Execution Trace:

1. `A = 00H` (Lower byte), `D = 00H` (Upper byte).
2. `B = 05H` (Multiplicand), `C = 03H` (Multiplier).

3. **Iteration 1 (C=3):**
   A = A + B $\Longrightarrow$ 00H + 05H = 05H. No carry.
   Decrement C $\Longrightarrow$ C = 02H.

4. **Iteration 2 (C=2):**
   A = A + B $\Longrightarrow$ 05H + 05H = 0AH. No carry.
   Decrement C $\Longrightarrow$ C = 01H.

5. **Iteration 3 (C=1):**
   A = A + B $\Longrightarrow$ 0AH + 05H = 0FH. No carry.
   Decrement C $\Longrightarrow$ C = 00H.

6. Loop ends because C reaches 0.

Result: Product is 000FH. Lower byte is 0FH, upper byte is 00H.

2.3 Data Transfer and Manipulation

Transferring blocks of data and finding specific elements (like the largest or smallest number) are common array operations.

Methodology:

Block Transfer of Data To move a block of data from a source memory location to a destination location:

1. Initialize the source memory pointer (e.g. HL pair) to the starting address of the source block.
2. Initialize the destination memory pointer (e.g. DE pair) to the starting address of the destination block.
3. Initialize a register (e.g. Register C) as a counter with the number of bytes to transfer.
4. Read a byte from the source memory into the Accumulator.
5. Write the byte from the Accumulator to the destination memory.
6. Increment both source and destination pointers.
7. Decrement the counter.
8. If the counter is not zero jump back to step 4.

Worked Example:

Transfer a block of 3 bytes Transfer 3 bytes of data from starting address 3000H to 4000H.
Program:

Address	Mnemonics	Comments
2000	LXI H, 3000H	Initialize source pointer HL = 3000H.
2003	LXI D, 4000H	Initialize destination pointer DE = 4000H.
2006	MVI C, 03H	Initialize counter C = 03H.
2008	MOV A, M	**LOOP:** Load data from source into A.
2009	STAX D	Store A into destination address pointed by DE.
200A	INX H	Increment source pointer.
200B	INX D	Increment destination pointer.
200C	DCR C	Decrement counter.
200D	JNZ 2008H	If counter $\neq 0$ jump to LOOP.
2010	HLT	Halt program.

The loop runs 3 times sequentially transferring bytes from 3000H, 3001H, and 3002H to 4000H, 4001H, and 4002H respectively.

Methodology:

Finding the Largest Element in an Array To find the largest number in a given array of data:

1. Initialize a memory pointer (HL pair) to the start of the array.
2. Load the length of the array from memory into a counter register (e.g. Register C).
3. Decrement the counter by 1 (since the first element is initially assumed to be the largest).
4. Increment the memory pointer and load the first array element into the Accumulator.
5. Increment the memory pointer to point to the next element.
6. Compare the Accumulator with the memory element (`CMP M`).
7. If the Accumulator is smaller (Carry flag is set) update the Accumulator with the memory element (`MOV A, M`).
8. Decrement the counter.
9. If the counter is not zero jump back to step 5.
10. Store the final Accumulator value (the largest number) in the designated memory location.

Worked Example:

Find the largest number among 15H 45H 12H Find the largest number in an array of 3 elements. The array length is stored at 2000H and elements at 2001H-2003H. Store result at 2004H. Data: [2000H]=03H, [2001H]=15H, [2002H]=45H, [2003H]=12H.

Execution Trace:

1. Pointer `HL = 2000H`. Counter `C = [2000H] = 03H`.
2. Decrement `C`. `C = 02H`.
3. Increment `HL = 2001H`. Load Accumulator `A = [2001H] = 15H`.
4. **First Iteration (C=2):**
   Increment `HL = 2002H`. Compare `A` (15H) with `M` (45H).
   15H − 45H generates a borrow (CY=1). Thus `A` < `M`.
   Since CY=1, update `A`: `MOV A, M` $\implies$ `A = 45H`.
   Decrement `C`. `C = 01H`.
5. **Second Iteration (C=1):**
   Increment `HL = 2003H`. Compare `A` (45H) with `M` (12H).
   45H − 12H does not generate a borrow (CY=0). Thus `A` $\geq$ `M`.
   Skip updating `A`.
   Decrement `C`. `C = 00H`.
6. **Loop ends.** `C` is 0.
7. Store `A` (45H) at 2004H.

The largest value 45H is successfully found and stored.

CHAPTER 3

Microcontroller Architecture

3.1 Determining the State of Program Status Word Flags

The Program Status Word (PSW) register contains status flags that reflect the result of arithmetic and logical operations in the 8051 microcontroller.

Methodology:

Determining PSW Flag Status To find the state of the Carry (CY) Auxiliary Carry (AC) Overflow (OV) and Parity (P) flags after an 8-bit addition:

1. Convert both 8-bit hexadecimal numbers to binary.
2. Perform binary addition bit by bit starting from the Least Significant Bit (D0) to the Most Significant Bit (D7).
3. **Carry Flag (CY):** Set to 1 if there is a carry out from the MSB (D7). Otherwise 0.
4. **Auxiliary Carry Flag (AC):** Set to 1 if there is a carry from D3 to D4. Otherwise 0.
5. **Overflow Flag (OV):** Set to 1 if there is a carry out of D6 but no carry out of D7 OR a carry out of D7 but no carry out of D6. It indicates a signed arithmetic overflow.
6. **Parity Flag (P):** Set to 1 if the number of 1s in the accumulator (result) is odd. Set to 0 if the number of 1s is even (even parity).

Worked Example:

Calculating PSW Flags after Addition Calculate the result and the state of the CY AC OV and P flags when the 8051 adds the numbers `38H` and `2FH`.

Step 1: Convert hex numbers to binary.

- `38H` = `0011 1000`
- `2FH` = `0010 1111`

Step 2: Perform binary addition.

$$\begin{array}{rll} & \texttt{0011 1000} & (\texttt{38H}) \\ + & \texttt{0010 1111} & (\texttt{2FH}) \\ \hline & \texttt{0110 0111} & (\texttt{67H}) \end{array}$$

Step 3: Evaluate flags based on the result.

- **CY (Carry Flag):** There is no carry out from D7. Therefore CY = 0.
- **AC (Auxiliary Carry):** Adding D3 bits (1 + 1) yields 0 with a carry of 1 to D4. Therefore AC = 1.
- **OV (Overflow Flag):** There is no carry out from D6 and no carry out from D7. Therefore OV = 0.
- **P (Parity Flag):** The result `0110 0111` contains five 1s (odd number). Therefore P = 1.

3.2 Stack Operations and Stack Pointer Calculation

The Stack Pointer (SP) points to the top of the stack. In the 8051 the SP is 8-bit wide and increments before storing data (PUSH) and decrements after retrieving data (POP). The default value of SP upon reset is `07H`.

Methodology:

Calculating SP during PUSH and POP Operations

1. **Initial State:** Identify the current value of the Stack Pointer (SP).
2. **PUSH Operation:**
   - Increment SP: SP $\leftarrow$ SP + 1.
   - Store the 1-byte data at the internal RAM address pointed to by the new SP.
3. **POP Operation:**
   - Retrieve the 1-byte data from the internal RAM address pointed to by the current SP.
   - Decrement SP: SP $\leftarrow$ SP $-$ 1.

Worked Example:

Tracking SP and Stack Memory Assume the SP is initially at `07H`. The CPU executes the following instructions:

```
MOV R0, #25H
MOV R1, #3AH
PUSH 00H    ; PUSH R0 onto stack
PUSH 01H    ; PUSH R1 onto stack
POP 02H     ; POP into R2
```

Determine the value of SP and the contents of the stack memory after each PUSH and POP instruction.
Step 1: Analyze PUSH 00H.

- Instruction pushes the content of R0 (`25H`) onto the stack.
- SP is incremented: SP = `07H` + 1 = `08H`.
- Memory location `08H` now contains `25H`.

Step 2: Analyze PUSH 01H.

- Instruction pushes the content of R1 (`3AH`) onto the stack.
- SP is incremented: SP = `08H` + 1 = `09H`.
- Memory location `09H` now contains `3AH`.

Step 3: Analyze POP 02H.

- Instruction pops the top of the stack into register R2 (address `02H`).
- Data at current SP (`09H`) which is `3AH` is copied to R2.
- SP is decremented: SP = `09H` $-$ 1 = `08H`.

Final State: SP = `08H` RAM location `08H` = `25H` RAM location `09H` = `3AH` R2 = `3AH`.

3.3 Timer Reload Value Calculation for Delay Generation

To generate a specific time delay using 8051 timers we must calculate the exact number of machine cycles required and load the timers with the appropriate initial values.

Methodology:

Calculating Timer Reload Values

1. **Calculate Machine Cycle Frequency:** Divide the oscillator crystal frequency (f_{osc}) by 12.

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

2. **Calculate Machine Cycle Time (Time Period):** Take the reciprocal of the machine cycle frequency.

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

3. **Calculate Required Number of Cycles (N):** Divide the desired time delay (T_{delay}) by the machine cycle time.

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

4. **Calculate Reload Value (Mode 1 - 16-bit Timer):**

$$\text{Reload Value} = 65536 - N$$

Convert the decimal reload value to 16-bit hexadecimal. The higher byte goes to THx and the lower byte goes to TLx.

5. **Calculate Reload Value (Mode 2 - 8-bit Auto-reload):**

$$\text{Reload Value} = 256 - N$$

Convert the decimal reload value to 8-bit hexadecimal and load it into THx.

Worked Example:

Calculating THx and TLx for a 5 ms Delay Calculate the values to be loaded into Timer 0 (Mode 1) to generate a delay of 5 ms. Assume the oscillator frequency is 11.0592 MHz.

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

$$f_{osc} = 11.0592 \text{ MHz} = 11,059,200 \text{ Hz}$$

$$T_{mc} = \frac{12}{11,059,200} = 1.085 \times 10^{-6} \text{ seconds} = 1.085 \text{ ţs}$$

Step 2: Calculate Number of Cycles (N).

$$T_{delay} = 5 \text{ ms} = 5000 \text{ ţs}$$

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

Step 3: Calculate the 16-bit Reload Value.

$$\text{Reload Value} = 65536 - 4608 = 60928$$

Step 4: Convert to Hexadecimal and Assign to Registers.

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

- Higher Byte: TH0 = EEH
- Lower Byte: TL0 = 00H

3.4 Baud Rate Calculation for Serial Communication

The 8051 microcontroller relies on Timer 1 operating in Mode 2 (8-bit auto-reload) to generate the baud rate for serial communication (UART Mode 1 and 3).

Methodology:

Calculating TH1 for a Specific Baud Rate The baud rate in UART Mode 1 and 3 is determined by the timer overflow rate. The Double Baud Rate bit (SMOD) in the PCON register can optionally double the baud rate.

1. Use the standard formula for baud rate:

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

2. Rearrange the formula to solve for TH1:

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

3. For standard operations assume SMOD = 0 (so $2^{\text{SMOD}} = 1$). If the problem states "Double Baud Rate is enabled" use SMOD = 1 ($2^{\text{SMOD}} = 2$).

4. Calculate the decimal value for TH1 and convert it to its hexadecimal equivalent.

Worked Example:

Finding TH1 for 9600 Baud Rate Find the TH1 value needed to configure the serial port for a baud rate of 9600 bps. Assume a crystal frequency of 11.0592 MHz and SMOD = 0.

Step 1: Identify given parameters.

- $f_{osc} = 11,059,200$ Hz
- Baud Rate = 9600 bps
- SMOD = 0 ($2^{\text{SMOD}} = 1$)

Step 2: Apply the TH1 calculation formula.

$$\text{TH1} = 256 - \left(\frac{1 \times 11,059,200}{384 \times 9600} \right)$$

Step 3: Simplify the expression.

$$\text{TH1} = 256 - \left(\frac{11,059,200}{3,686,400} \right)$$

$$\text{TH1} = 256 - 3 = 253$$

Step 4: Convert to hexadecimal.

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

Therefore to achieve a baud rate of 9600 bps Timer 1 register **TH1** must be loaded with **FDH**.

3.5 Determining Internal RAM and Bit Addresses

The 8051 has 128 bytes of internal RAM which is divided into Register Banks (00H-1FH) Bit-Addressable RAM (20H-2FH) and General Purpose RAM (30H-7FH).

Methodology:

Finding Register and Bit Addresses

1. **Register Bank Addresses:**
   - Bank 0: Registers R0 to R7 are mapped to RAM addresses `00H` to `07H`.
   - Bank 1: Registers R0 to R7 are mapped to RAM addresses `08H` to `0FH`.
   - Bank 2: Registers R0 to R7 are mapped to RAM addresses `10H` to `17H`.
   - Bank 3: Registers R0 to R7 are mapped to RAM addresses `18H` to `1FH`.
2. **Formula for Register Address:** For register R_n in Bank B the direct RAM address is:

$$\text{Address} = (B \times 8) + n$$

3. **Bit-Addressable RAM:** The RAM bytes from `20H` to `2FH` provide 128 bit-addressable locations numbered `00H` to `7FH`.
4. **Formula for Bit Address:** For bit k ($0 \leq k \leq 7$) of RAM byte M (`20H` $\leq M \leq$ `2FH`):

$$\text{Bit Address} = ((M - 20_{16}) \times 8) + k$$

   Note: Perform the subtraction and multiplication in decimal then convert the final result back to hex.

Worked Example:

Calculating RAM and Bit Addresses (a) Find the direct RAM address of Register R5 in Bank 2.
(b) Find the exact bit address of bit 4 in RAM byte location `2BH`.
Solution for (a): Address of R5 in Bank 2

- Bank $B = 2$ Register $n = 5$.
- Address $= (2 \times 8) + 5 = 16 + 5 = 21$ (in decimal).
- Convert decimal 21 to hexadecimal: $21 = 16 + 5 \rightarrow$ `15H`.
- Alternatively using hex directly: Bank 2 starts at `10H`. 10H + 5 = `15H`.

Solution for (b): Bit address of bit 4 in RAM byte 2BH

- Byte Address $M =$ `2BH` $= 43_{10}$.
- Bit number $k = 4$.
- Bit Address $= ((43 - 32) \times 8) + 4 = (11 \times 8) + 4 = 88 + 4 = 92_{10}$.
- Convert decimal 92 to hexadecimal: $92/16 = 5$ with remainder 12 (which is C_{16}).
- The bit address is **5CH**.

3.6 Calculating Available Memory for Interrupt Service Routines

The 8051 has fixed vector addresses for its hardware interrupts. When an interrupt occurs the Program Counter (PC) jumps to the corresponding vector address. If multiple interrupts are used the space between vector addresses dictates whether the Interrupt Service Routine (ISR) can be placed directly or if a jump instruction is required.

Methodology:

Analyzing Interrupt Vector Space The standard interrupt vector table for the 8051 is:

Interrupt Source	Vector Address
External Interrupt 0 (INT0)	`0003H`
Timer 0 Overflow (TF0)	`000BH`
External Interrupt 1 (INT1)	`0013H`
Timer 1 Overflow (TF1)	`001BH`
Serial Port (RI / TI)	`0023H`
Timer 2 (8052 only)	`002BH`

1. Identify the vector address of the target interrupt.
2. Identify the vector address of the next consecutive interrupt in memory.
3. **Calculate Available Space:** Subtract the target address from the next address.

$$\text{Available Bytes} = \text{Next Vector Address} - \text{Target Vector Address}$$

4. If the ISR requires more bytes than available a `LJMP` (Long Jump) instruction to a different memory location must be placed at the vector address.

Worked Example:

Determining Space for Timer 0 ISR A programmer is writing an Interrupt Service Routine (ISR) for Timer 0. The program also uses External Interrupt 1 (`INT1`). Determine the maximum number of bytes available for the Timer 0 ISR before it overlaps with the `INT1` vector address.

Step 1: Identify the vector addresses.

- Timer 0 Vector Address = `000BH`
- External Interrupt 1 (`INT1`) Vector Address = `0013H`

Step 2: Calculate the difference in hexadecimal.

- Subtract `000BH` from `0013H`.
- In decimal: `13H` $= 19_{10}$ and `0BH` $= 11_{10}$.
- Difference $= 19 - 11 = 8$ bytes.

Conclusion: There are exactly **8 bytes** of memory available between the Timer 0 vector and the `INT1` vector. If the Timer 0 ISR requires more than 8 bytes the programmer must place a jump instruction (e.g. `LJMP MAIN_ISR`) at address `000BH` that redirects execution to a larger memory area.

CHAPTER 4

8051 Programming

4.1 Addressing Modes

The 8051 microcontroller provides various mechanisms to access operands known as addressing modes. Choosing the correct addressing mode is essential for efficient data manipulation.

Methodology:

Determination of Operand Address 1. **Immediate Addressing**: The operand is a constant value explicitly given in the instruction. It is denoted by the `#` symbol. 2. **Register Addressing**: The operand is stored in one of the eight working registers (`R0` to `R7`) of the currently selected register bank or the Accumulator (`A`). 3. **Direct Addressing**: The 8-bit address of the memory location is specified directly in the instruction. It is used to access internal RAM and Special Function Registers (SFRs). 4. **Indirect Addressing**: A register (`R0` or `R1`) holds the address of the operand. It is indicated by the `@` symbol. 5. **Indexed Addressing**: The address is formed by adding an offset in the Accumulator (`A`) to a base pointer (`DPTR` or `PC`). It is typically used for reading lookup tables from ROM. 6. **Relative Addressing**: The operand is an 8-bit signed offset added to the Program Counter (`PC`) to determine the target branch address. 7. **Bit Addressing**: Specific instructions operate on single bits in bit-addressable RAM or SFRs.

Worked Example:

Identify the Addressing Mode and Operand Identify the addressing mode for the following 8051 instructions and explain their operation: 1. `MOV A, #45H` 2. `MOV R2, A` 3. `MOV 30H, A` 4. `MOV A, @R0` 5. `MOVC A, @A+DPTR`

Solution:

Instruction	Addressing Mode	Operation Description
`MOV A, #45H`	Immediate	Loads the literal hex value 45H into the Accumulator.
`MOV R2, A`	Register	Copies the content of the Accumulator into register R2.
`MOV 30H, A`	Direct	Stores the Accumulator content into internal RAM address 30H.
`MOV A, @R0`	Indirect	Moves data from the RAM address held in R0 to the Accumulator.
`MOVC A, @A+DPTR`	Indexed	Reads a code byte from ROM at address A + DPTR into A.

4.2 Data Transfer and Stack Operations

Methodology:

Block Data Transfer Algorithm To move an array of data bytes from a source memory block to a destination memory block: 1. Initialize the source memory pointer using a register (e.g. `R0`). 2. Initialize the destination memory pointer using another register (e.g. `R1`). 3. Load the total number of bytes to transfer into a loop counter (e.g. `R2`). 4. Read a byte from the source address using indirect addressing (`MOV A, @R0`). 5. Write the byte to the destination address using indirect addressing (`MOV @R1, A`). 6. Increment both the source

pointer (`INC R0`) and destination pointer (`INC R1`). 7. Decrement the loop counter and branch back to step 4 if not zero (`DJNZ R2, loop_label`).

Worked Example:

Block Transfer in Internal RAM Write an 8051 assembly language program to transfer 5 bytes of data starting from internal RAM address `40H` to a block starting at `50H`.
Solution:

```
      MOV R0, #40H      ; Initialize R0 as source pointer
      MOV R1, #50H      ; Initialize R1 as destination pointer
      MOV R2, #05H      ; Load the block size (5 bytes) into counter R2
AGAIN:
      MOV A, @R0        ; Read data byte from source into Accumulator
      MOV @R1, A        ; Store data byte into destination
      INC R0            ; Increment source pointer
      INC R1            ; Increment destination pointer
      DJNZ R2, AGAIN    ; Decrement counter and jump to AGAIN if not zero
      SJMP $            ; Halt the program execution
```

Methodology:

Stack Operation Algorithm The stack is a Last-In-First-Out (LIFO) memory used for temporary data storage and preserving contexts. 1. **Push Operation**: - Increment the Stack Pointer (`SP`) by 1. - Store the direct memory byte at the new address pointed to by `SP`. 2. **Pop Operation**: - Retrieve the byte from the address pointed to by `SP`. - Decrement the Stack Pointer (`SP`) by 1.

Worked Example:

Register Exchange Using the Stack Write a program to exchange the contents of internal RAM addresses `30H` and `40H` using only stack operations.
Solution:

```
      PUSH 30H          ; Push the content of address 30H onto the stack
      PUSH 40H          ; Push the content of address 40H onto the stack
      POP 30H           ; Pop the top of stack (originally 40H) into 30H
      POP 40H           ; Pop the next item (originally 30H) into 40H
      SJMP $            ; Halt the program
```

4.3 Arithmetic Instructions

Methodology:

Multibyte Addition Algorithm Since the 8051 is an 8-bit microcontroller, adding 16-bit numbers requires splitting the operation into two 8-bit additions. 1. Clear the carry flag before starting (`CLR C`). 2. Load the lower order byte of the first number into the Accumulator. 3. Add the lower order byte of the second number using `ADD A, operand`. 4. Store the resulting lower byte of the sum. 5. Load the higher order byte of the first number into the Accumulator. 6. Add the higher order byte of the second number using `ADDC A, operand` (Add with Carry) to incorporate any carry generated from the lower byte addition. 7. Store the resulting higher byte of the sum. 8. If needed, capture the final carry bit.

Worked Example:

16-Bit Number Addition Write a program to add two 16-bit numbers. The first number is stored at 30H (Low Byte) and 31H (High Byte). The second number is stored at 32H (Low Byte) and 33H (High Byte). Store the result at 34H (Low Byte) and 35H (High Byte).
Solution:

```
MOV A, 30H          ; Load lower byte of 1st number into A
ADD A, 32H          ; Add lower byte of 2nd number
MOV 34H, A          ; Store lower byte of the result

MOV A, 31H          ; Load higher byte of 1st number into A
ADDC A, 33H         ; Add higher byte of 2nd number along with carry
MOV 35H, A          ; Store higher byte of the result
SJMP $              ; Halt the program
```

Methodology:

Multiplication and Division Algorithm 1. **Multiplication**: - Load the first 8-bit unsigned number into Accumulator A. - Load the second 8-bit unsigned number into Register B. - Execute the MUL AB instruction. - The 16-bit result is stored across B (High Byte) and A (Low Byte). 2. **Division**: - Load the 8-bit dividend into Accumulator A. - Load the 8-bit divisor into Register B. - Execute the DIV AB instruction. - The quotient is stored in A and the remainder is stored in B.

Worked Example:

Arithmetic Computations using MUL and DIV Write an 8051 assembly snippet to multiply the number 15H by 0AH and store the result in registers R3 (high byte) and R4 (low byte).
Solution:

```
MOV A, #15H         ; Load multiplicand into A
MOV B, #0AH         ; Load multiplier into B
MUL AB              ; Multiply A and B (Result: A=D2H, B=00H)
MOV R4, A           ; Store lower byte of result in R4
MOV R3, B           ; Store higher byte of result in R3
```

4.4 Logical Instructions and Data Masking

Methodology:

Algorithm for Bit Masking and Data Unpacking Masking involves clearing or setting specific bits within a byte while leaving others unchanged. 1. To **isolate the lower nibble** (bits 0-3): Perform a logical AND (ANL) operation between the data byte and the mask 0FH. 2. To **isolate the upper nibble** (bits 4-7): - Perform a nibble swap using the SWAP A instruction (moves upper nibble to lower nibble position). - Perform a logical AND (ANL) with the mask 0FH. 3. Store the unpacked digits in their respective memory locations.

Worked Example:

Unpack a BCD Number A packed BCD number is stored at memory location 40H (e.g., 95H). Write an assembly program to unpack this number and store the lower BCD digit at 41H (e.g., 05H) and the upper BCD digit at 42H (e.g., 09H).
Solution:

```
MOV A, 40H          ; Read the packed BCD byte (e.g., 95H)
MOV R1, A           ; Backup the original byte in R1
```

```
        ; Extract lower nibble
        ANL A, #0FH         ; Mask upper nibble (A = 05H)
        MOV 41H, A          ; Store lower unpacked digit at 41H

        ; Extract upper nibble
        MOV A, R1           ; Restore original byte (95H) into A
        SWAP A              ; Swap nibbles (A = 59H)
        ANL A, #0FH         ; Mask upper nibble (A = 09H)
        MOV 42H, A          ; Store upper unpacked digit at 42H

        SJMP $              ; Halt execution
```

4.5 Branching and Conditional Programming

Methodology:

Array Evaluation and Branching Algorithm To evaluate a condition across an array (e.g., finding the maximum value): 1. Initialize a pointer to the array's starting address. 2. Initialize a counter with the size of the array minus one. 3. Load the first array element into a reference register (e.g., Accumulator). 4. Increment the pointer and read the next element. 5. Compare the two values using `CJNE` (Compare and Jump if Not Equal). - In 8051, `CJNE A, operand, rel` sets the Carry flag (`C=1`) if `A` $<$ `operand`. - It clears the Carry flag (`C=0`) if `A` $\geq$ `operand`. 6. Use `JC` (Jump if Carry) or `JNC` (Jump if No Carry) to conditionally update the reference register. 7. Decrement the counter and repeat from step 4 until all elements are processed. 8. Store the final evaluated result.

Worked Example:

Find the Largest Number in an Array Write an 8051 assembly language program to find the largest unsigned number in a block of 5 data bytes stored starting at internal RAM address `50H`. Store the largest number at address `60H`.

Solution:

```
        MOV R0, #50H        ; Pointer to start of the array
        MOV R2, #04H        ; Loop counter = N - 1 = 4 comparisons
        MOV A, @R0          ; Load the first element into A (Assume it is max)

NEXT:   INC R0              ; Point to the next element
        MOV B, @R0          ; Load the next element into B
        CJNE A, B, CHK      ; Compare A and B
CHK:    JNC SKIP            ; If A >= B, Carry = 0. Jump to SKIP
        MOV A, B            ; If A < B, Carry = 1. Update A with larger value

SKIP:   DJNZ R2, NEXT       ; Decrement counter, repeat if not zero
        MOV 60H, A          ; Store the maximum value found at 60H
        SJMP $              ; Halt the program
```

Methodology:

Algorithm for Counting Bits To count occurrences of a specific binary state, such as the number of logic 1s in a byte: 1. Load the byte into the Accumulator. 2. Initialize a bit counter register to 0. 3. Initialize a loop counter to 8 (since there are 8 bits in a byte). 4. Rotate the Accumulator left or right through the carry flag (using `RLC A` or `RRC A`). 5. Check the carry flag using `JNC` (Jump if No Carry). 6. If the carry flag is set, increment the bit counter. 7. Decrement the loop counter and repeat from step 4 until all 8 bits are checked.

Worked Example:

Count Number of 1s in a Data Byte Write an assembly program to count the number of 1s in the data byte stored at memory address `30H` and save the total count at address `31H`.

Solution:

```
      MOV A, 30H          ; Load data byte into Accumulator
      MOV R1, #00H        ; Initialize bit count register R1 to 0
      MOV R2, #08H        ; Loop counter for 8 bits

BITCHK:
      RRC A               ; Rotate Accumulator right through carry
      JNC NOT_ONE         ; If Carry is 0, skip incrementing
      INC R1              ; If Carry is 1, increment the bit counter

NOT_ONE:
      DJNZ R2, BITCHK     ; Decrement loop counter and repeat 8 times
      MOV 31H, R1         ; Store the total count of 1s at address 31H
      SJMP $              ; Halt program
```

CHAPTER 5

Interfacing and Microcontroller Systems

5.1 Display and Output Interfacing

Methodology:

Hex Code Generation for Seven Segment Display A 7-segment display consists of 7 LEDs (labeled a through g) and an optional decimal point (dp). The microcontroller port pins are mapped to these segments (e.g. P1.0 to a, P1.1 to b, ..., P1.7 to dp).
Steps to generate the Hex Code: 1. Determine the Display Type: - Common Cathode (CC): Segments are turned ON by sending a Logic 1. - Common Anode (CA): Segments are turned ON by sending a Logic 0. 2. Identify the active segments required to form the desired numeric character. 3. Form an 8-bit binary number (dp-g-f-e-d-c-b-a) where each bit corresponds to a segment. - For CC: Assign 1 to active segments and 0 to inactive ones. - For CA: Assign 0 to active segments and 1 to inactive ones. 4. Convert the 8-bit binary number into a two-digit Hexadecimal value.

Worked Example:

Determine Hex Codes for Digits 2 and 5 Find the hex codes to display the decimal digits '2' and '5' on both Common Cathode and Common Anode 7-segment displays. Assume the mapping is: dp=Bit7, g=Bit6, f=Bit5, e=Bit4, d=Bit3, c=Bit2, b=Bit1, a=Bit0.
Solution: Step 1: Determine active segments. For digit '2': Active segments are a, b, d, e, g. Inactive: c, f, dp. For digit '5': Active segments are a, c, d, f, g. Inactive: b, e, dp.
Step 2: Common Cathode (CC) Codes (1 = ON, 0 = OFF) For digit '2': - dp=0, g=1, f=0, e=1, d=1, c=0, b=1, a=1 - Binary: 0101 1011 → Hex: 5BH For digit '5': - dp=0, g=1, f=1, e=0, d=1, c=1, b=0, a=1 - Binary: 0110 1101 → Hex: 6DH
Step 3: Common Anode (CA) Codes (0 = ON, 1 = OFF) (Note: The CA code is the 1's complement of the CC code). For digit '2': - Binary: 1010 0100 → Hex: A4H For digit '5': - Binary: 1001 0010 → Hex: 92H

Methodology:

Algorithm for LCD Command and Data Write To display information on a standard 16x2 alphanumeric LCD (such as HD44780) the microcontroller must send commands (to configure the LCD) and data (the ASCII characters to display).
Command Write Sequence (RS = 0): 1. Place the 8-bit command code on the LCD data pins. 2. Set RS = 0 (Register Select for Command). 3. Set R/W = 0 (Write mode). 4. Send a High-to-Low pulse on the Enable (E) pin (E = 1, delay, E = 0) to latch the command. 5. Provide sufficient delay for the LCD to process the command.
Data Write Sequence (RS = 1): 1. Place the 8-bit ASCII data on the LCD data pins. 2. Set RS = 1 (Register Select for Data). 3. Set R/W = 0 (Write mode). 4. Send a High-to-Low pulse on the Enable (E) pin (E = 1, delay, E = 0) to latch the data. 5. Provide a short delay for the LCD to process the data.

Worked Example:

LCD Sequence for Displaying a Character Assuming the LCD is connected to Port 1 (Data) and Port 2 (Control: P2.0=RS, P2.1=R/W, P2.2=E). Specify the step-by-step logic levels on these ports to display the character 'A' on a cleared display.
Solution: Step 1: Clear Display Command - Command for Clear Display is 01H. - Set Port 1 = 01H. - Set RS = 0 (P2.0 = 0) and R/W = 0 (P2.1 = 0). - Pulse E: Set E = 1 (P2.2 = 1) then wait and Set E = 0 (P2.2 = 0).
Step 2: Display Character 'A' - The ASCII value for 'A' is 41H. - Set Port 1 = 41H. - Set RS = 1 (P2.0 = 1) and R/W = 0 (P2.1 = 0). - Pulse E: Set E = 1 (P2.2 = 1) then wait and Set E = 0 (P2.2 = 0).

5.2 Digital to Analog Converter Interfacing

Methodology:

Calculating DAC0808 Output Voltage The DAC0808 is an 8-bit Digital-to-Analog Converter. It converts an 8-bit digital input into an analog output current which is typically converted to an analog voltage (V_{out}) using an operational amplifier.
Steps to calculate Output Voltage: 1. Identify the Reference Voltage (V_{ref}). 2. Calculate the Step Size (Resolution):

$$\text{Step Size} = \frac{V_{ref}}{2^n}$$

For an 8-bit DAC ($n = 8$), Step Size $= \frac{V_{ref}}{256}$. 3. Convert the 8-bit Digital Input to its equivalent Decimal value (D). 4. Calculate the Analog Output Voltage:

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

Worked Example:

Calculate DAC0808 Analog Output A DAC0808 is interfaced with an 8051 microcontroller. The reference voltage V_{ref} is set to 5V. Calculate the analog output voltage for the following digital inputs sent by the microcontroller: (a) 00H (b) 80H (c) C8H and (d) FFH.
Solution: Step 1: Calculate the Step Size.

$$\text{Step Size} = \frac{5\text{V}}{256} = 0.01953\text{V} \approx 19.53\text{mV}$$

Step 2: Calculate Output Voltage for each Input. (a) For Input = 00H: - Decimal value $D = 0$. - $V_{out} = 19.53\text{mV} \times 0 = 0\text{V}$.
(b) For Input = 80H: - Decimal value $D = 128$. - $V_{out} = 19.53\text{mV} \times 128 = 2.5\text{V}$.
(c) For Input = C8H: - Decimal value $D = 12 \times 16 + 8 = 192 + 8 = 200$. - $V_{out} = 19.53\text{mV} \times 200 = 3.906\text{V}$.
(d) For Input = FFH: - Decimal value $D = 255$. - $V_{out} = 19.53\text{mV} \times 255 = 4.98\text{V}$.

Methodology:

Algorithm for Waveform Generation using DAC By continuously sending a sequence of digital values to the DAC specific analog waveforms can be generated.
Square Wave Generation: 1. Send the maximum value (FFH) to the DAC port. 2. Call a delay subroutine (determines the HIGH time). 3. Send the minimum value (00H) to the DAC port. 4. Call a delay subroutine (determines the LOW time). 5. Repeat continuously.
Triangular Wave Generation: 1. Initialize a register with 00H. 2. Increment the register and output it to the DAC port until it reaches FFH (Rising Edge). 3. Decrement the register and output it to the DAC port until it reaches 00H (Falling Edge). 4. Repeat continuously.

Worked Example:

Generating a Triangular Wave Sequence Trace the sequence of hexadecimal values generated by the microcontroller and calculate the corresponding analog voltages (assume $V_{ref} = 5V$ and Step Size = 19.53mV) for the first 3 steps of the rising edge and the first 3 steps of the falling edge of a triangular wave.
Solution: Rising Edge (Starting from 00H): - Step 1: Output = 00H ($D = 0$) $\rightarrow V_{out} = 0 \times 19.53\text{mV} = 0\text{V}$ - Step 2: Output = 01H ($D = 1$) $\rightarrow V_{out} = 1 \times 19.53\text{mV} = 19.53\text{mV}$ - Step 3: Output = 02H ($D = 2$) $\rightarrow V_{out} = 2 \times 19.53\text{mV} = 39.06\text{mV}$
Falling Edge (Starting from FFH): - Step 1: Output = FFH ($D = 255$) $\rightarrow V_{out} = 255 \times 19.53\text{mV} = 4.98\text{V}$ - Step 2: Output = FEH ($D = 254$) $\rightarrow V_{out} = 254 \times 19.53\text{mV} = 4.96\text{V}$ - Step 3: Output = FDH ($D = 253$) $\rightarrow V_{out} = 253 \times 19.53\text{mV} = 4.94\text{V}$

5.3 Analog to Digital Converter Interfacing

Methodology:

Calculating ADC0804 Digital Output The ADC0804 is an 8-bit Analog-to-Digital Converter. It converts an unknown analog input voltage (V_{in}) into an 8-bit digital value.
Steps to calculate Digital Output: 1. Identify the Full-Scale Reference Voltage (V_{ref}). For ADC0804 this is determined by the $V_{ref}/2$ pin. (e.g. if $V_{ref}/2 = 2.5V$ then $V_{ref} = 5.0V$). 2. Calculate the Step Size (Resolution):

$$\text{Step Size} = \frac{V_{ref}}{2^n} = \frac{V_{ref}}{256}$$

3. Calculate the Digital Equivalent (D) in decimal:

$$D = \frac{V_{in}}{\text{Step Size}}$$

4. Round D to the nearest integer and convert it to Hexadecimal.

Worked Example:

Calculate Digital Output of ADC0804 An ADC0804 is configured such that its $V_{ref}/2$ pin is supplied with $1.28V$. Calculate the step size and the final digital output (in Hexadecimal) for analog input voltages of (a) $1.5V$ and (b) $2.1V$.
Solution: Step 1: Determine Full-Scale Voltage and Step Size. - Full-scale Voltage $V_{ref} = 2 \times 1.28\text{V} = 2.56\text{V}$. - Step Size = $2.56\text{V}/256 = 0.01\text{V} = 10\text{mV}$.
Step 2: Calculate Digital Output for $V_{in} = 1.5V$. - $D = \frac{V_{in}}{\text{Step Size}} = \frac{1.5\text{V}}{0.01\text{V}} = 150$. - Decimal 150 in Hexadecimal is 96H ($150/16 = 9$ remainder 6). - Digital Output = 96H.
Step 3: Calculate Digital Output for $V_{in} = 2.1V$. - $D = \frac{V_{in}}{\text{Step Size}} = \frac{2.1\text{V}}{0.01\text{V}} = 210$. - Decimal 210 in Hexadecimal is D2H ($210/16 = 13$ remainder 2). - Digital Output = D2H.

Methodology:

Algorithm for Reading Data from ADC0804 The ADC0804 must be properly controlled by the microcontroller to initiate a conversion and retrieve the result. The primary control pins are CS (Chip Select) WR (Start Conversion) INTR (End of Conversion) and RD (Read Data).
Steps for Interfacing: 1. Make CS = 0 to select the ADC chip. 2. Send a Low-to-High pulse on the WR pin (WR = 0 wait WR = 1) to start the analog-to-digital conversion. 3. Monitor the INTR (Interrupt) pin. Wait in a loop as long as INTR = 1. 4. When INTR = 0 the conversion is complete. 5. Make RD = 0 to bring the digital data out of the ADC registers onto the data bus. 6. Read the data from the connected microcontroller port. 7. Make RD = 1 and CS = 1 to end the read operation.

Worked Example:

ADC Read Sequence Timing Assuming the ADC is connected with Control Pins to Port 3 (P3.0 = CS, P3.1 = WR, P3.2 = RD, P3.3 = INTR) and Data Pins to Port 1. Write the sequence of logical states applied to Port 3 to successfully read a sample.
Solution: 1. **Chip Select:** Set P3.0 = 0. 2. **Start Conversion:** Set P3.1 = 0 then P3.1 = 1 (Pulse WR). 3. **Wait for EOC:** Check P3.3. If P3.3 == 1 keep checking. When P3.3 == 0 proceed. 4. **Read Data:** Set P3.2 = 0 to enable the output buffer of the ADC. 5. **Fetch Data:** Read Port 1 into an internal register (e.g. Accumulator). 6. **Reset:** Set P3.2 = 1 (disable read) and P3.0 = 1 (deselect chip).

5.4 Room Temperature Controller System

Methodology:

Calculating ADC Values for LM35 Temperature Sensor The LM35 is an analog temperature sensor that provides an output voltage directly proportional to the temperature in degrees Celsius (°C). It is commonly interfaced with an ADC0804 to build digital temperature controllers.
Steps for Calibration and Calculation: 1. Calculate Sensor Voltage (V_{LM35}): The LM35 has a scale factor of 10mV/°C.

$$V_{LM35} = \text{Temperature in } ^\circ\text{C} \times 10\text{mV}$$

2. Match ADC Resolution for Direct Display: If the ADC step size is set exactly equal to the sensor's scale factor (10mV), then the decimal value of the ADC output will identically match the temperature in °C. To achieve a step size of 10mV set the ADC $V_{ref}/2$ pin to $1.28V$ (since $2.56V/256 = 10\text{mV}$). 3. Calculate Final Digital Value:

$$D = \frac{V_{LM35}}{\text{Step Size}}$$

Worked Example:

Calculate Digital Output for a Room Temperature Controller A temperature controller uses an LM35 sensor interfaced with an ADC0804. The ADC's $V_{ref}/2$ pin is configured for $1.28V$. (a) Calculate the analog voltage generated by the LM35 at 35°C. (b) Calculate the digital output of the ADC (in Hexadecimal) at this temperature. (c) If the microcontroller is programmed to turn ON a cooling fan relay when the temperature exceeds 40°C, what hexadecimal threshold value should the microcontroller compare the input against?
Solution: Part (a): Analog Voltage at 35°C - V_{LM35} = Temperature × 10mV = 35 × 10mV = 350mV = 0.35V.
Part (b): Digital Output of ADC - Step Size = (2 × 1.28V)/256 = 2.56V/256 = 0.01V = 10mV. - Digital Output $D = \frac{V_{LM35}}{\text{Step Size}} = \frac{350\text{mV}}{10\text{mV}} = 35$. - Decimal 35 in Hexadecimal = 23H (35/16 = 2 remainder 3). - The ADC output is 23H.
Part (c): Threshold Value for 40°C - Analog voltage at 40°C = 40 × 10mV = 400mV. - Decimal Threshold = 400mV/10mV = 40. - Decimal 40 in Hexadecimal = 28H (40/16 = 2 remainder 8). - The microcontroller must compare the incoming ADC reading with the threshold value of 28H.

5.5 Push Button Switch and Relay Interfacing

Methodology:

Algorithm for Push Button Debouncing When a mechanical push button is pressed its contacts may bounce causing multiple spurious transitions before settling. A microcontroller must use software debouncing to avoid interpreting a single press as multiple presses.
Steps for Software Debouncing: 1. Read the pin connected to the switch. 2. If an active state is detected (e.g. Logic 0 for an active-low switch) wait for a small delay (typically 10ms to 20ms). 3. Read the pin again.

4. If the pin is still in the active state consider it a valid button press. Otherwise ignore it as electrical noise. 5. Wait for the pin to return to its inactive state before accepting the next press (Wait for Button Release) to prevent repeated triggering.

Worked Example:

Switch Press Detection Sequence A microcontroller reads an active-low push button switch connected to Port 2.0. Write the logical sequence of operations to detect a valid press turn ON a Relay (active-high) connected to Port 2.1 and turn it OFF when the switch is released.

Solution: 1. **Initial Check:** Read P2.0. If P2.0 == 1 stay in a loop checking P2.0 continuously. 2. **Debounce Delay:** If P2.0 == 0 call a 20ms delay subroutine. 3. **Verify Press:** Read P2.0 again. - If P2.0 == 1 the press was noise. Return to the Initial Check. - If P2.0 == 0 the press is validated. 4. **Activate Output:** Set P2.1 = 1 (Turn ON Relay). 5. **Wait for Release:** Read P2.0 in a loop until P2.0 == 1. 6. **Debounce Release:** Call a 20ms delay subroutine. 7. **Verify Release:** Read P2.0. If P2.0 == 1 set P2.1 = 0 (Turn OFF Relay) and return to the Initial Check.

Appendix: Key Formulae

This appendix provides a quick reference to the general formulae and mathematical relationships discussed in this book.

Unit 1: Microprocessor Fundamentals

- **Clock Period & Frequency:**

$$T = \frac{1}{f}$$
$$f = \frac{1}{T}$$

- **Instruction Execution Time:**

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

where N_T is the total number of T-states for the instruction.

- **Memory Capacity:**

$$\text{Total Locations} = 2^N$$

where N is the number of address lines.

- **Memory Address Range:**

$$\text{Ending Address} = \text{Starting Address} + \text{Size} - 1$$
$$\text{Total Size} = \text{Ending Address} - \text{Starting Address} + 1$$

- **CPU Performance:**

$$\text{Instructions Per Second} = \frac{f}{\text{CPI}}$$
$$\text{MIPS} = \frac{\text{Instructions Per Second}}{10^6}$$

where CPI is Cycles Per Instruction and MIPS is Millions of Instructions Per Second.

Unit 2: Assembly Language Programming

- **Delay Loop Execution Time:**

$$T_{delay} = T \times (T_{init} + N \times T_{loop} - 3)$$

where T_{init} is the initialization time, T_{loop} is the time per loop iteration, and N is the number of iterations.

Unit 3: 8051 Microcontroller Architecture

- **Machine Cycle Frequency & Time:**

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

- **Bit-Addressable RAM Address Calculation:**

$$\text{Bit Address} = ((M - 20_{16}) \times 8) + k$$

  where M is the byte address (20_{16} to $2F_{16}$) and k is the bit position (0 to 7).

- **Timer Operations:**

$$N = \frac{T_{delay}}{T_{mc}}$$
$$\text{8-bit Timer Reload Value} = 256 - N$$
$$\text{16-bit Timer Reload Value} = 65536 - N$$

- **Serial Communication (Baud Rate):**

$$\text{Baud Rate} = \frac{2^{\text{SMOD}}}{32} \times \frac{f_{osc}}{12 \times (256 - \text{TH1})}$$
$$\text{TH1 Reload Value} = 256 - \left(\frac{2^{\text{SMOD}} \times f_{osc}}{384 \times \text{Baud Rate}}\right)$$

- **Interrupt Vector Space:**

$$\text{Available Bytes} = \text{Next Vector Address} - \text{Target Vector Address}$$

Unit 5: Interfacing with 8051

- **Data Converters (ADC/DAC):**

$$\text{Step Size} = \frac{V_{ref}}{2^n}$$
$$V_{out}\ (\text{DAC}) = \text{Step Size} \times D$$
$$D\ (\text{ADC Digital Output}) = \frac{V_{in}}{\text{Step Size}}$$

  where n is the resolution in bits, V_{ref} is the reference voltage, and D is the digital value.

- **LM35 Temperature Sensor:**

$$V_{LM35} = \text{Temperature in } ^\circ\text{C} \times 10 \text{ mV}$$