Experiment No. 4: Introduction to 8086 Microprocessor - Architecture and Addressing Modes

1. Aim

To introduce the 8086 microprocessor architecture, understand its segmented memory organization, and practically demonstrate various addressing modes used for accessing data.

2. Objectives

Upon completion of this experiment, the student will be able to:

- **Identify and describe the functional units and register organization of the 8086 microprocessor.**
- **Explain the concept of segmented memory and calculate physical addresses.**
- **Differentiate between various 8086 addressing modes.**
- **Write and execute 8086 assembly language programs using different addressing modes.**
- **Analyze the execution flow of programs and observe how addressing modes affect data access.**

3. Theory

The Intel 8086 is a 16-bit microprocessor, a significant advancement over the 8-bit 8085. It features a 20-bit address bus, enabling it to access 220 or 1 Megabyte (1 MB) of memory. Its 16-bit data bus allows it to fetch or store 16 bits of data at a time. A key architectural innovation in the 8086 is its segmented memory.

3.1. 8086 Architecture and Register Organization

The 8086 architecture is divided into two main units, enabling pipelining for improved performance:

- **Bus Interface Unit (BIU): Handles all external bus operations like fetching instructions, reading/writing data, and I/O operations. It contains:**
  - **Segment Registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment). These are 16-bit registers used for memory segmentation.**
  - **Instruction Pointer (IP): A 16-bit register holding the offset address of the next instruction within the current code segment.**
  - **Address Generation Unit: Responsible for calculating the 20-bit physical memory address.**
  - **Instruction Queue: A 6-byte FIFO (First-In, First-Out) buffer that pre-fetches instructions, enabling pipelining.**

- **Execution Unit (EU): Decodes and executes instructions. It contains:**
    - **General Purpose Registers: Eight 16-bit registers (AX, BX, CX, DX, SP, BP, SI, DI). These can also be accessed as 8-bit registers (e.g., AH/AL for AX).**
        - **AX (Accumulator): Used for arithmetic, logic, and I/O operations.**
        - **BX (Base Register): Often used as a base address for memory access.**
        - **CX (Count Register): Primarily used as a loop counter.**
        - **DX (Data Register): Used for I/O operations, multiply/divide, and holds the most significant part of a 32-bit product/dividend.**
        - **SP (Stack Pointer): Points to the top of the stack within the stack segment.**
        - **BP (Base Pointer): Used to access data on the stack.**
        - **SI (Source Index): Used as an index for source data in string operations.**
        - **DI (Destination Index): Used as an index for destination data in string operations.**
    - **Flags Register: A 16-bit register containing various condition flags (e.g., Zero, Carry, Sign, Overflow, Parity) and control flags (e.g., Direction, Interrupt, Trap).**
    - **Arithmetic Logic Unit (ALU): Performs arithmetic and logical operations.**

3.2. Segmented Memory Organization

The 8086's 1 MB physical memory is divided into logical segments. Each segment can be up to 64 KB in size and starts at an address divisible by 16 (a paragraph boundary). This segmentation allows for flexible memory management, protection, and allows a 16-bit register to access a larger memory space.

Physical Address Calculation:

A 20-bit physical address is generated by combining a 16-bit segment address (from a segment register) and a 16-bit offset address (from a general-purpose register, pointer, or index register, or immediate value).

The formula for physical address calculation is:

Physical Address = (Segment Register Value * 10H) + Offset Address

Or, equivalently:

Physical Address = (Segment Register Value << 4) + Offset Address

Here, `* 10H` or `<< 4` (left shift by 4 bits) effectively appends four 0s to the right of the segment register value, making it a 20-bit base address, which is then added to the 16-bit offset.

Numerical Example:

If DS (Data Segment) = 2000H and the offset address for a variable is 0050H.

Physical Address = (2000Htimes10H) + 0050H

Physical Address = 20000H+0050H

Physical Address = 20050H

This means the actual memory location accessed is 20050H.

The default segment register for different types of memory access are:

- **Code Segment (CS): Points to the segment where the currently executing program instructions are located. Offset is provided by IP (Instruction Pointer).**
- **Data Segment (DS): Points to the segment where most data is stored. Default offset from general purpose registers like BX, SI, DI, or direct addresses.**
- **Stack Segment (SS): Points to the segment where the program stack is located. Default offset from SP (Stack Pointer) or BP (Base Pointer).**
- **Extra Segment (ES): An additional data segment, often used for string operations.**

3.3. Addressing Modes

Addressing modes define how the effective address of an operand is calculated. The 8086 supports various addressing modes, providing flexibility and efficiency in accessing data.

1. **Immediate Addressing Mode:**
    - **The operand is a part of the instruction itself. No memory access is required for the operand.**
    - **Syntax: `MOV AX, 5000H` (Loads the value 5000H directly into AX).**
    - **Use Case: Loading constant values into registers.**
2. **Register Addressing Mode:**
    - **The operand is located in one of the 8086's internal registers.**
    - **Syntax: `MOV AX, BX` (Copies the content of BX into AX).**
    - **Use Case: Fast data transfer between registers, arithmetic/logic operations on register data.**
3. **Direct Addressing Mode:**
    - **The operand's effective address (offset) is directly specified as part of the instruction. The physical address is calculated using DS as the segment register.**
    - **Syntax: `MOV AX, [1234H]` (Copies the word from memory location DS:1234H into AX).**
    - **Use Case: Accessing fixed memory locations, global variables.**
4. **Register Indirect Addressing Mode:**
    - **The effective address of the operand is held in one of the pointer or index registers (BX, BP, SI, DI). The default segment register is DS for BX, SI, DI, and SS for BP.**
    - **Syntax: `MOV AX, [BX]` (Copies the word from memory location DS:BX into AX).**

    - **Use Case: Accessing data from an array or a list where the starting address is in a register and the offset changes.**
5. **Based Addressing Mode:**
    - **The effective address is calculated by adding a displacement (8-bit or 16-bit) to the contents of a base register (BX or BP).**
    - **Syntax: `MOV AX, [BX + 10H]` or `MOV AX, 10H[BX]` (Copies the word from DS: (BX + 10H) into AX).**
    - **Use Case: Accessing elements of a record or structure, where BX points to the base of the record and 10H is the offset of a specific field.**
6. **Indexed Addressing Mode:**
    - **The effective address is calculated by adding a displacement (8-bit or 16-bit) to the contents of an index register (SI or DI).**
    - **Syntax: `MOV AX, [SI + 20H]` or `MOV AX, 20H[SI]` (Copies the word from DS: (SI + 20H) into AX).**
    - **Use Case: Similar to based addressing, used for accessing array elements or structures.**
7. **Based-Indexed Addressing Mode:**
    - **The effective address is calculated by adding the contents of a base register (BX or BP) and an index register (SI or DI), plus an optional displacement.**
    - **Syntax: `MOV AX, [BX + SI]` or `MOV AX, [BX + SI + 30H]` (Copies the word from DS: (BX + SI + 30H) into AX).**
    - **Use Case: Accessing elements in two-dimensional arrays, where BX might hold the base address of a row and SI holds the offset within the row.**
8. **String Addressing Mode:**
    - **Used by string instructions (e.g., MOVS, CMPS, SCAS, LODS, STOS). SI is implicitly used for the source address (DS:SI) and DI for the destination address (ES:DI). After each operation, SI and DI are automatically incremented or decremented based on the Direction Flag (DF).**
    - **Syntax: `MOVSB` (Moves a byte from DS:SI to ES:DI).**
    - **Use Case: Efficient block memory transfers.**

4. Materials Required

- **8086 Microprocessor Trainer Kit (if available)**
- **OR 8086 Simulator Software (e.g., MASM/TASM with DEBUG or emu8086, DOSBox with MASM/LINK/DEBUG)**
- **Personal Computer**

5. Procedure

Part A: Familiarization with 8086 Architecture (Using Simulator/Trainer Kit)

1. **Launch the 8086 Simulator (e.g., emu8086): Open the chosen simulator.**
2. **Identify Registers: Locate the display areas for the 8086's internal registers (AX, BX, CX, DX, SP, BP, SI, DI, CS, DS, SS, ES, IP, Flags).**

3. **Observe Memory View: Access the memory view window. Note how memory addresses are typically displayed in segment:offset format (e.g., 1000:0100).**
4. **Observe Code View: Locate the area where assembly instructions are displayed.**
5. **Run a Sample Program (Optional): Load a very simple program (e.g., `MOV AX, 0005H; HLT`) and single-step through it to observe how the Instruction Pointer (IP) and register contents change.**

Part B: Programs Demonstrating Addressing Modes

For each addressing mode, write the assembly code, assemble/compile it, and execute it using the simulator/trainer kit. Observe the changes in registers and memory.

Pre-requisite: Initialize some memory locations with known values for testing. For instance, store values starting from DS:1000H.

Example:

ORG 1000H

DB 01H, 02H, 03H, 04H, 05H, 06H, 07H, 08H

DW 1234H, 5678H, 9ABCH, DEF0H

---

1. Immediate Addressing Mode

- **Aim: Load immediate values into registers.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
   ; No data defined for this example as values are immediate
.CODE
MAIN PROC FAR
   MOV AX, 4C00H   ; Exit to DOS (standard termination for .COM/.EXE)
   INT 21H

   MOV AX, 1234H   ; Load immediate word 1234H into AX
   MOV BL, 56H     ; Load immediate byte 56H into BL
   MOV CX, 789AH   ; Load immediate word 789AH into CX
   HLT             ; Halt instruction (or use standard exit)
MAIN ENDP
END MAIN
```

-
-

- **Execution & Observation:**
    - **Set breakpoints after each `MOV` instruction or single-step.**
    - **Observe AX register content becomes `1234H`.**
    - **Observe BL register content becomes `56H`.**
    - **Observe CX register content becomes `789AH`.**

2. Register Addressing Mode

- **Aim: Transfer data between registers.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
.CODE
MAIN PROC FAR
   MOV AX, 4C00H
   INT 21H

   MOV AX, 0001H   ; Initialize AX with 0001H
   MOV BX, 0002H   ; Initialize BX with 0002H
   MOV CX, AX      ; Copy content of AX to CX (CX becomes 0001H)
   ADD DX, BX      ; Add content of BX to DX (DX = DX + 0002H)
   HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
    - **After `MOV CX, AX`, observe CX becomes `0001H`.**
    - **After `ADD DX, BX`, observe DX is incremented by `0002H`.**

3. Direct Addressing Mode

- **Aim: Access data from a specific memory location using its direct offset.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
   MY_VAR WORD 1234H ; Define a word variable initialized with 1234H
   MY_BYTE BYTE 56H  ; Define a byte variable initialized with 56H
.CODE
```

```
MAIN PROC FAR
    MOV AX, 4C00H
    INT 21H

    MOV AX, @DATA   ; Load segment address of data segment into AX
    MOV DS, AX      ; Initialize DS with the data segment address

    MOV BX, MY_VAR  ; Load content of MY_VAR (1234H) into BX
                    ; Equivalent to MOV BX, [OFFSET MY_VAR]

    MOV CL, MY_BYTE ; Load content of MY_BYTE (56H) into CL
                    ; Equivalent to MOV CL, [OFFSET MY_BYTE]
    HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
  - **After `MOV DS, AX`, observe DS holds the correct data segment address.**
  - **After `MOV BX, MY_VAR`, observe BX becomes `1234H`.**
  - **After `MOV CL, MY_BYTE`, observe CL becomes `56H`.**
  - **Verify the physical addresses accessed: For `MY_VAR`, it would be DS:OFFSET(MY_VAR).**

4. Register Indirect Addressing Mode

- **Aim: Access data using the address stored in a register.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
    MY_ARRAY WORD 10H, 20H, 30H ; Define an array of words
.CODE
MAIN PROC FAR
    MOV AX, 4C00H
    INT 21H

    MOV AX, @DATA
    MOV DS, AX      ; Initialize DS

    MOV BX, OFFSET MY_ARRAY ; Load the offset address of MY_ARRAY into BX
    MOV AX, [BX]    ; Load word from DS:BX (10H) into AX
    INC BX          ; Increment BX by 1 (points to next byte)
    INC BX          ; Increment BX by 1 (points to next word)
    MOV CX, [BX]    ; Load word from DS:BX (20H) into CX
```

```
    HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
  - **After `MOV BX, OFFSET MY_ARRAY`, observe BX holds the offset of `MY_ARRAY`.**
  - **After `MOV AX, [BX]`, observe AX becomes `0010H`.**
  - **After the two `INC BX` instructions, BX will point to the next word element.**
  - **After `MOV CX, [BX]`, observe CX becomes `0020H`.**
  - **Note: For word (16-bit) access, the offset increments by 2.**

5. Based Addressing Mode

- **Aim: Access data using a base register (BX/BP) plus a displacement.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
    MY_RECORD LABEL WORD
    FIELD1 WORD 1111H
    FIELD2 WORD 2222H
    FIELD3 BYTE 33H
.CODE
MAIN PROC FAR
    MOV AX, 4C00H
    INT 21H

    MOV AX, @DATA
    MOV DS, AX      ; Initialize DS

    MOV BX, OFFSET MY_RECORD ; BX points to the start of MY_RECORD

    MOV AX, [BX + 0]    ; Load FIELD1 (1111H) into AX (displacement 0)
    MOV DX, [BX + 2]    ; Load FIELD2 (2222H) into DX (displacement 2 bytes from start)
    MOV CL, [BX + 4]    ; Load FIELD3 (33H) into CL (displacement 4 bytes from start)
    HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
  - **After `MOV AX, [BX + 0]`, observe AX becomes `1111H`.**

- **After `MOV DX, [BX + 2]`, observe DX becomes `2222H`. (Since FIELD1 is a word, FIELD2 starts 2 bytes after FIELD1).**
- **After `MOV CL, [BX + 4]`, observe CL becomes `33H`. (FIELD1 and FIELD2 are 2 bytes each, so FIELD3 starts 4 bytes after MY_RECORD).**

6. Indexed Addressing Mode

- **Aim: Access data using an index register (SI/DI) plus a displacement.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
   GRADE_ARRAY BYTE 90, 85, 70, 95, 60
.CODE
MAIN PROC FAR
   MOV AX, 4C00H
   INT 21H

   MOV AX, @DATA
   MOV DS, AX      ; Initialize DS

   MOV SI, 0       ; SI points to the first element (offset 0)

   MOV AL, GRADE_ARRAY[SI + 0] ; Load 90 into AL (direct indexed)
   MOV BL, [SI + 1]            ; Load 85 into BL (offset 1 from base address)
   MOV CL, [SI + 3]            ; Load 95 into CL (offset 3 from base address)
   HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
  - **After `MOV AL, GRADE_ARRAY[SI + 0]`, observe AL becomes `90H`.**
  - **After `MOV BL, [SI + 1]`, observe BL becomes `85H`.**
  - **After `MOV CL, [SI + 3]`, observe CL becomes `95H`.**

7. Based-Indexed Addressing Mode

- **Aim: Access data using a base register (BX/BP) and an index register (SI/DI), with an optional displacement.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
   MATRIX LABEL WORD ; Represents a 2x3 matrix of words
   ROW1 DW 10H, 20H, 30H
   ROW2 DW 40H, 50H, 60H
.CODE
MAIN PROC FAR
   MOV AX, 4C00H
   INT 21H

   MOV AX, @DATA
   MOV DS, AX      ; Initialize DS

   MOV BX, OFFSET MATRIX   ; BX points to the start of the matrix (e.g., ROW1)
   MOV SI, 2 * 2          ; SI = 4 (offset to the start of ROW2 if each word is 2 bytes and row 1 has 2 words)
                          ; For a 2x3 matrix of words, a row has 3 words = 6 bytes.
                          ; So, SI for ROW2 would be 6 (offset from MATRIX base).
                          ; Let's assume we want to access ROW2, element 1 (50H)
   MOV BX, OFFSET MATRIX   ; BX points to the start of the MATRIX (ROW1)
   MOV SI, 6              ; SI represents the offset to the beginning of ROW2 (3 words * 2 bytes/word = 6 bytes)
   MOV DI, 2              ; DI represents the offset to the 2nd element within a row (1 word * 2 bytes/word = 2 bytes)

   MOV AX, [BX + SI + DI]  ; Access element at MATRIX[ROW2_OFFSET + ELEMENT_OFFSET]
                          ; This would access MATRIX + 6 + 2 = MATRIX + 8
                          ; Which is the 5th word (index 4) if counting from 0 (MATRIX[0]=10H, MATRIX[1]=20H, MATRIX[2]=30H, MATRIX[3]=40H, MATRIX[4]=50H)
                          ; So, AX will get 50H.
   HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
  - **After `MOV AX, [BX + SI + DI]`, observe AX becomes `0050H`.**
  - **Explain how the physical address (DS:BX+SI+DI) is calculated by the CPU.**

8. String Addressing Mode (Illustrative)

- **Aim: Demonstrate basic string operation with implicit addressing.**
- **Assembly Code:**
- **Code snippet**

```
.MODEL SMALL
.STACK 100H
.DATA
   SOURCE_STR DB 'HELLO', 0 ; Source string, null-terminated
   DEST_STR   DB 10 DUP(?)  ; Destination buffer
.CODE
MAIN PROC FAR
   MOV AX, 4C00H
   INT 21H

   MOV AX, @DATA
   MOV DS, AX      ; Initialize DS
   MOV ES, AX      ; Initialize ES (for destination)

   MOV SI, OFFSET SOURCE_STR   ; SI points to source string
   MOV DI, OFFSET DEST_STR     ; DI points to destination string

   CLD             ; Clear Direction Flag (DF=0, auto-increment SI, DI)
   MOV CX, 6       ; Number of bytes to move (5 letters + null terminator)
   REP MOVSB       ; Repeat Move String Byte until CX is zero

   HLT
MAIN ENDP
END MAIN
```

-
-
- **Execution & Observation:**
  - **Before `REP MOVSB`, observe content of `DEST_STR` (random).**
  - **After `REP MOVSB`, observe `DEST_STR` contains 'HELLO' followed by a null byte.**
  - **Observe SI and DI increment by 6 (for 6 bytes moved).**
  - **Observe CX becomes `0000H`.**

6. Observations

Record detailed observations for each program executed. This should include:

- **Initial register values.**
- **Memory contents at relevant addresses (before and after operations).**
- **Register values after each key instruction (especially for single-stepping).**
- **Any changes in flags (if applicable and visible in simulator).**
- **The final state of registers and memory to confirm the desired operation.**

Example Format for Observations (for each Addressing Mode Program):

Program: [Addressing Mode Name]

- **Initial State:**
  - **DS: [Value]**

    - **AX: [Value]**
    - **BX: [Value]**
    - **CL: [Value]**
    - **Memory at [Relevant Address]: [Content] (e.g., 1000H:01H02H03Hdots)**
- **Step-by-Step Execution & Changes:**
    - **Instruction: `MOV AX, 1234H`**
        - **Observation: AX changes from [Previous Value] to `1234H`.**
        - **Explanation: [Brief explanation of what happened, e.g., "Immediate value 1234H was loaded into the Accumulator."]**
    - **Instruction: `MOV BX, MY_VAR`**
        - **Observation: BX changes from [Previous Value] to `1234H`. Memory location for MY_VAR (e.g., DS:0000H) was `34H 12H`.**
        - **Explanation: "The word at the direct memory address corresponding to MY_VAR was loaded into BX."**
    - **... (Continue for critical instructions)**
- **Final State:**
    - **AX: [Final Value]**
    - **BX: [Final Value]**
    - **CL: [Final Value]**
    - **Memory at [Relevant Address]: [Final Content]**

7. Deliverables

1. **Explanation of 8086 Architecture: (As covered in Theory, Section 3.1).**
2. **Explanation of Segmented Memory and Physical Address Calculation with Numerical Example: (As covered in Theory, Section 3.2).**
3. **Explanation of Each Addressing Mode: (As covered in Theory, Section 3.3).**
4. **Assembly Code with Comments: For each addressing mode demonstration program (as provided in Section 5, Part B).**
5. **Execution Screenshots from Simulator: Include screenshots showing:**
    - **Initial state of registers and relevant memory areas.**
    - **State after key instructions or the final state after program execution, clearly demonstrating the effect of the addressing mode.**
    - **(Optional but recommended) Screenshot of physical address calculation for a specific instruction if the simulator provides this debug feature.**
6. **Detailed Observations and Analysis: (As recorded in Section 6).**

8. Conclusion

This experiment successfully provided an introduction to the 8086 microprocessor's architecture and its innovative segmented memory organization. We learned how 20-bit physical addresses are generated from 16-bit segment and offset values. Furthermore, by writing and executing assembly language programs on a simulator, we gained a practical understanding of various 8086 addressing modes, including immediate, register, direct, register indirect, based, indexed, and based-indexed addressing. The analysis of program execution and observation of register and

memory changes clarified how each addressing mode influences data access, which is fundamental to efficient 8086 programming.
