Chapter 2

Addressing Modes

Data Declaration

In computer organization and assembly language, when we work with data, we need a way to declare and store that data for our programs. Here's a simple explanation:

1. **Immediate Operand**:
    - In assembly language, we can include data directly within instructions. For example, in the instruction **"mov ax, 5"**, the number **5** is part of the instruction itself. This is known as an **immediate operand**.
2. **Data in Memory**:
    - However, real programs often need to work with data that's stored in memory, not just constants. Data in memory can be changed, unlike constants. So, we need a way to store and retrieve data from memory.
3. **Assembler Directives**:
    - To declare a part of our program as holding data, we use special commands called assembler directives. Two basic directives are **"define byte" (written as "db")** and **"define word" (written as "dw").** These directives allow us to reserve memory space to store data.
    - For example, with **"db somevalue"**, we reserve a cell in memory with the value "somevalue" in it, which we can use in our program. "dw" is used for reserving larger memory space, typically **16 bits** or **2 bytes**.
4. **Labels**:
    - To work with this data later in the program, we need to know where it's stored in memory. This is where labels come in. A label is like a symbol that we associate with a memory address.
    - We create labels in our code by giving a name followed by a colon, like **"my_variable:"**. We can use this label to refer to the data.

Direct Addressing

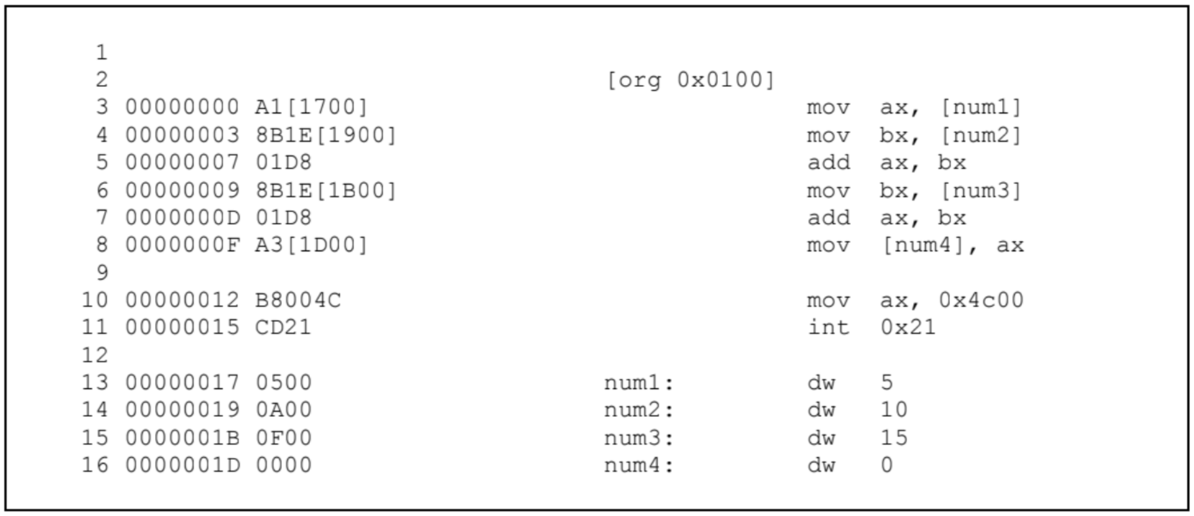
Adding Three Numbers Using Memory Variables:

```
[org 0x0100]
mov ax, [num1]     ; Load the first number in ax
mov bx, [num2]     ; Load the second number in bx
add ax, bx         ; Accumulate the sum in ax
mov bx, [num3]     ; Load the third number in bx
add ax, bx         ; Accumulate the sum in ax
mov [num4], ax     ; Store the sum in num4
mov ax, 0x4c00     ; Terminate the program
int 0x21
```

```
num1: dw 5
num2: dw 10
num3: dw 15
num4: dw 0
```

In this example:

- `num1`, `num2`, `num3`, and `num4` are labels associated with memory addresses. They make it easier to reference specific memory locations in the code.
- Labels are followed by a colon (`:`) to distinguish them from instructions.
- `mov ax, [num1]`: This instruction uses direct addressing mode. It loads the value stored in memory at the address pointed to by `num1` into the `ax` register. The square brackets `[...]` indicate that the data is in memory.



```

                                [org 0x0100]
00000000 A1[1700]                            mov   ax, [num1]
00000003 8B1E[1900]                          mov   bx, [num2]
00000007 01D8                                add   ax, bx
00000009 8B1E[1B00]                          mov   bx, [num3]
0000000D 01D8                                add   ax, bx
0000000F A3[1D00]                            mov   [num4], ax

00000012 B8004C                              mov   ax, 0x4c00
00000015 CD21                                int   0x21

00000017 0500                   num1:        dw    5
00000019 0A00                   num2:        dw    10
0000001B 0F00                   num3:        dw    15
0000001D 0000                   num4:        dw    0
```

We're examining the listing file generated after assembling the program. Here's a simplified explanation:

1. **Opcode Change**:
   - In our program, the first instruction used to be represented by the opcode B80500.
   - The opcode **B8** is used for moving constants into the AX register.
   - However, the new instruction is represented by the opcode A11700.
   - Opcode **A1** is used when moving data from memory into the AX register.
   - **1700**, which can also be expressed as **0017** in hexadecimal represents an offset in memory, specifically the offset of a label called **"num1"** in our code.
2. **Assembler's Calculation**:
   - The assembler calculates the offset of the **"num1"** label and replaces references to "num1" throughout the program.

- For instance, when we see "mov ax, [num1]" in the code, it is essentially moving data from memory location **0017** to **AX**.

3. **Memory Contents**:
   - At offset 0017 in the file, we find the value 0500.
   - So, the contents of memory location 0017 are interpreted as the number 0005 when stored as a word.
4. **Debugger Operation**:
   - When we load the program into a debugger, it's loaded at offset **0100** in memory.
   - This displacement affects all memory accesses in our program.
   - For instance, the instruction **"A11700"** becomes **"A11701"**, meaning that our variable is now located at offset 0117 in memory.
5. **Debugger Behavior**:
   - It's important to note that the debugger may interpret our data as code and display it as meaningless instructions.
   - This happens because the debugger treats everything in the code window as code, and it cannot differentiate our declared data from opcodes.
   - We must ensure that we terminate execution before our data is unintentionally executed as code.
6. **Label Naming**:
   - Inside the debugger, the labels we used in our code, such as "num1," "num2," "num3," and "num4," are not displayed.
   - Instead, the debugger shows the corresponding memory addresses like **0117**, **0119**, **011B**, and **011D**.

Adding Three Numbers Using a Single Label:

We can write the same program using a single label. As we know that num2 is two ahead of num1, we can use num1+2 instead of num2 and let the assembler calculate the sum during assembly process.

```
[org 0x0100]
mov ax, [num1]      ; Load the first number in ax
mov bx, [num1+2]    ; Load the second number in bx
add ax, bx          ; Accumulate the sum in ax
mov bx, [num1+4]    ; Load the third number in bx
add ax, bx          ; Accumulate the sum in ax
mov [num1+6], ax    ; Store the sum at num1+6
mov ax, 0x4c00      ; Terminate the program
int 0x21

num1: dw 5
      dw 10
```

```
        dw 15
        dw 0
```

Labels are used extensively in this example to simplify memory addressing. Here, `num1` serves as a single label for multiple memory locations, making the code more concise and readable.

Declaring Data in a Single Line:

We do not need to place labels on individual variables we can save space and declare all data on a single line separated by commas.

```
[org 0x0100]
mov ax, [num1]      ; Load the first number in ax
mov bx, [num1+2]    ; Load the second number in bx
add ax, bx          ; Accumulate the sum in ax
mov bx, [num1+4]    ; Load the third number in bx
add ax, bx          ; Accumulate the sum in ax
mov [num1+6], ax    ; Store the sum at num1+6
mov ax, 0x4c00      ; Terminate the program
int 0x21

num1: dw 5, 10, 15, 0
```

Adding Three Numbers Directly in Memory:

```
[org 0x0100]
mov ax, [num1]      ; Load the first number in ax
mov [num1+6], ax    ; Store the first number in the result
mov ax, [num1+2]    ; Load the second number in ax
add [num1+6], ax    ; Add the second number to the result in memory
mov ax, [num1+4]    ; Load the third number in ax
add [num1+6], ax    ; Add the third number to the result in memory
mov ax, 0x4c00      ; Terminate the program
int 0x21

num1: dw 5, 10, 15, 0
```

In this example, we observe the use of direct addressing for both data loading and addition to memory. Instructions like `mov ax, [num1]` and `add [num1+6], ax` use direct addressing to manipulate data in memory.

The first two instructions of the last program read a number into AX and placed it at another memory location. A quick thought reveals that the following might be a possible single instruction to replace the couple.

mov [num1+6], [num1] ; ILLEGAL

However this form is illegal and not allowed on the Intel architecture. None of the general operations of mov add, sub etc. allow moving data from memory to memory. Only register to register, register to memory, memory to register, constant to memory, and constant to register operations are allowed.

Add three numbers using byte variables:

If we change the directive in the last example from **DW** to **DB**, the program will still assemble and debug without errors, however the results will not be the same as expected?

```
[org 0x0100]
mov ax, [num1]      ; Load the first number in ax
mov [num1+6], ax    ; Store the first number in the result
mov ax, [num1+2]    ; Load the second number in ax
add [num1+6], ax    ; Add the second number to the result in memory
mov ax, [num1+4]    ; Load the third number in ax
add [num1+6], ax    ; Add the third number to the result in memory
mov ax, 0x4c00      ; Terminate the program
int 0x21

num1: db 5, 10, 15, 0
```

Size Mismatch Errors:

Data Reading Issue:

- Data movement operations must match the size of the data involved.
- For instance, a word operation cannot fit into a byte-sized container.
- The assembler ensures that data movement instructions have matching sizes.
- When we read the first operand, it will be interpreted as **0A05** in the register.
- This happened because **DB** is used to allocate variable of **1 byte** while the registers **AX** takes input of **2 bytes** so two operands were placed in consecutive byte memory locations.

Logical Error Responsibility:

- This mismatch between data declaration and access is a logical error in the program.
- It's the programmer's responsibility to ensure that data declarations and their access are synchronized.

Data Type Declaration:

- In assembly, it's vital to access data with the correct data type (**byte** or **word**) as declared.
- The previous examples used **"DW"** for words, but if we intend to treat data as bytes, we should declare it differently.

```
[org 0x0100]
mov al, [num1]              ; load first number in al
mov bl, [num1+1]            ; load second number in bl
```

```
add al, bl                  ; accumulate sum in al
mov bl, [num1+2]            ; load third number in bl
add al, bl                  ; accumulate sum in al
mov [num1+3], al            ; store sum at num1+3
mov ax, 0x4c00              ; terminate program
int 0x21

num1: db 5, 10, 15, 0
```

Byte Variables:

- We modify the program to use byte variables instead of words.
- We load numbers into **AL** and **BL** registers, which are byte-sized registers.
- The data is accessed using byte offsets (e.g., **"num1+1"** instead of **"num1+2"**).
- We use the **"DB"** directive to declare data as bytes, ensuring that each data element occupies one byte.

mov byte [num1], 5
mov word [num1], 5

Register Indirect Addressing

- So far, we've been working with data one piece at a time. But what if we have many numbers to add? Doing it the way we did would require lots of instructions.
- To efficiently work with consecutive data in memory, we need a way to change the memory address dynamically.
- The solution is to use registers that can hold memory addresses, like **BX**, **BP**, **SI**, and **DI** in the **iAPX88** architecture.

Adding Three Numbers with Indirect Addressing:

```
[org 0x100]
mov bx, num1    ; Point BX to the first number
mov ax, [bx]    ; Load the first number into AX
add bx, 2       ; Advance BX to the second number
add ax, [bx]    ; Add the second number to AX
add bx, 2       ; Advance BX to the third number
add ax, [bx]    ; Add the third number to AX
add bx, 2       ; Advance BX to the result location
mov [bx], ax    ; Store the sum at num1+6
mov ax, 0x4c00  ; Terminate the program
int 0x21

num1: dw 5, 10, 15, 0 ; Define the numbers
```

Here, **BX** register acts as a pointer to our data, giving us control over which memory location to access.

An important thing in the above example is that a register is used to reference memory so this form of access is called register indirect memory access. We used the BX register for it and the B in BX and BP stands for base therefore we call register indirect memory access using **BX** or **BP**, **“based addressing”**. Similarly when **SI** or **DI** is used we name the method **“indexed addressing”**. They have the same functionality, with minor differences because of which the two are called base and index. The differences will be explained later.

Loops:

- To efficiently perform repetitive tasks, we need two things: the ability to change addresses dynamically and a way to repeat the same instruction.
- We introduce two new instructions for this purpose: `SUB` (subtract) and `JNZ` (jump if not zero).
- In this example, we'll add ten numbers, but this approach can be extended to handle any number of data elements.

Adding Ten Numbers with a Loop:

```
[org 0x0100]
mov bx, num1 ; Point BX to the first number
mov cx, 10 ; Load the count of numbers in CX
mov ax, 0 ; Initialize the sum to zero

l1: ; This is a code label, and it marks the beginning of a loop
add ax, [bx] ; Add the number to AX
add bx, 2 ; Advance BX to the next number
sub cx, 1 ; Reduce the count of numbers to be added
jnz l1 ; Jump to label l1 if CX is not zero

mov [total], ax ; Write the sum back in memory
mov ax, 0x4c00 ; Terminate the program
int 0x21

num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50 ; Define the numbers
total: dw 0 ; Initialize a variable to store the total
```

- You'll notice that the `JNZ` instruction (`jnz l1`) checks the zero flag **(ZF)** to decide whether to jump or not.
- The loop runs until **CX** becomes zero, which means it iterates ten times (as initially set).
- We use labels (`l1`) to mark where the program should jump, allowing us to create loops and control program flow.

Register + OFFSET Addressing

- In assembly language, we can use combinations of direct and indirect references for memory access.
- In this example, we'll use a single register (BX) to access different elements of an array by providing only the array index, not the exact address.

- This approach allows the same register to be used for accessing different arrays and for index comparison.

Adding Ten Numbers Using Register + Offset Addressing:

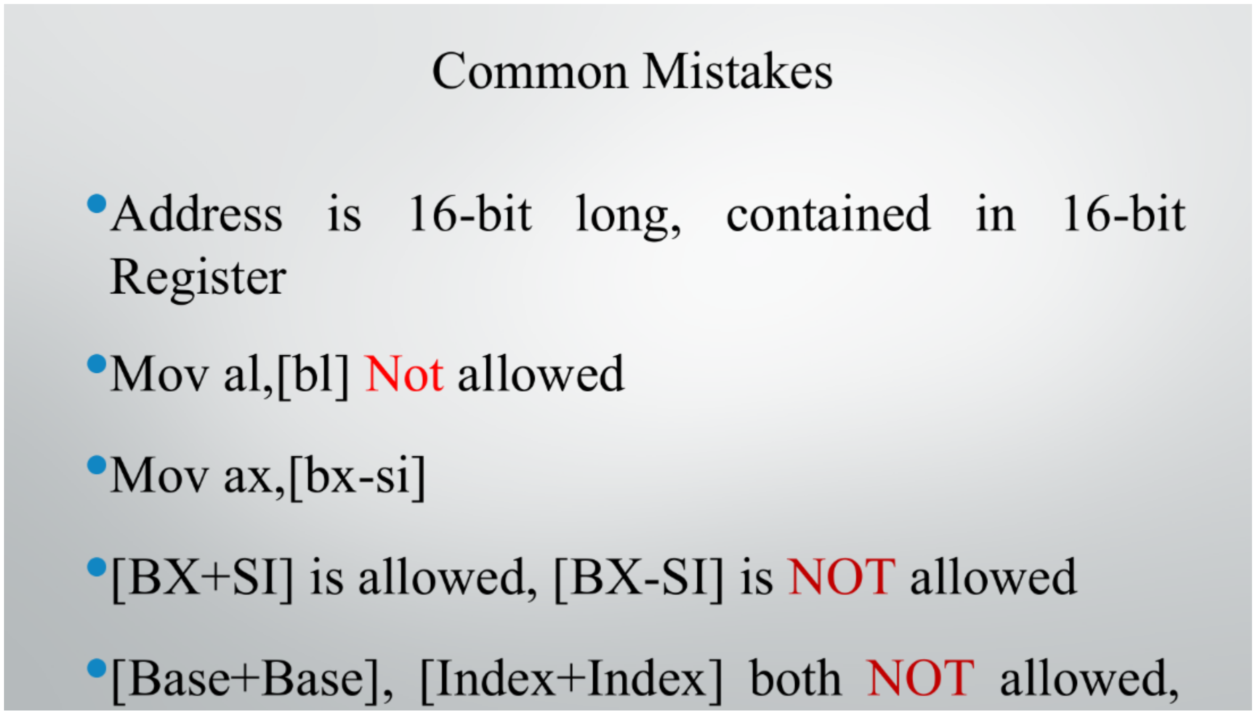
```
[org 0x0100]
mov bx, 0 ; Initialize array index to zero
mov cx, 10 ; Load count of numbers into CX
mov ax, 0 ; Initialize sum to zero

l1: ; Code label marking the start of a loop
add ax, [num1+bx] ; Add the number to AX (using register + offset)
add bx, 2 ; Advance BX to the next index
sub cx, 1 ; Reduce the count of numbers to be added
jnz l1 ; Jump to label l1 if CX is not zero

mov [total], ax ; Write the sum back in memory
mov ax, 0x4c00 ; Terminate the program
int 0x21

num1: dw 10, 20, 30, 40, 50, 10, 20, 30, 40, 50 ; Define the numbers
total: dw 0 ; Initialize a variable to store the total
```

This approach is known as **"base + offset"** or **"index + offset"** addressing, depending on whether BX or BP (or SI or DI) is used as the base register.



Common Mistakes

- Address is 16-bit long, contained in 16-bit Register
- Mov al,[bl] Not allowed
- Mov ax,[bx-si]
- [BX+SI] is allowed, [BX-SI] is NOT allowed
- [Base+Base], [Index+Index] both NOT allowed,