North South University

Assignment-01

Course: CSEl-331

Section: 07

Submitted To

Asif Ahmed Neloy

Submitted By

S.M Shouvik Islam

ID: 1712767642

Slide#1

Introduction

In this session, you will be introduced to assembly language programming and to emu8086 emulator software, emu8086 will be used as both an editor and as an assembler for all your assembly language programs

steps required to run an assembly program:

1. Write the necessary assembly source code.
2. Save the assembly source code.
3. ~~compl~~ compile/Assemble source code to create machine code.
4. Emulate/Run the machine code.

First, familiarize yourself with the software before you begin

to write any code. Follow the in-class instructions regarding the layout of emu 8086.

Micro controllers vs Microprocessors

- A microprocessor is a CPU on a single chip.
- If a microprocessor, its associated support circuitry, memory, and peripheral I/O components are implemented on a single chip, it is a micro controller.

Features of 8086

- 8086 is a 16 bit processor. It's ALU, internal registers work with 16 bit binary word.
- 8086 has a 16bit data bus. It can read or write data ~~or~~ to a memory /port either 16 bits

on 8 bits at a time.

- 8086 has a 20bit address bus which means, it can address up to $2^{20} = 1$ MB memory location.

Registrar - Register - Resistor

- Both ALU and FPU have a very small amount of super-fast private memory placed right next to them for their exclusive use. these are called registers.
- The ALU and FPU store intermediate and final results from their calculations in these registers
- Processed data goes back to the data cache and then to the main memory from these registers.

Inside the CPU : Get to know various registers

Registers are basically the CPU's own internal memory. they are used, among other purposes, to store temporary data while performing calculations. Let's look

at each one in details.

General purpose Registers (GPR)

The 8086 CPU has 8 general-purpose registers; each register has its own name:

- AX - The Accumulator register (divided into AH / AL).
- BX - The Base Address register (divided into BH / BL)
- CX - The count register (divided into CH / CL)
- DX - The Data register (divided into DH / DL)
- SI - Source Index Register
- DI - Destination Index register.
- BP - Base Pointer.
- SP - stack Pointer.

Despite the name of a register, it is the programmer who determines the usage for each general-purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bits.

4 general-purpose registers (AX, BX, CX, DX) are made of two separates 8 bit registers, for example if AX = 0011000000110010b then AH = 00110000b and AL = 00110010b. Therefore, when you modify any of the 8-bit

registers 16 bit registers are also updated, and vice-versa. The same is for other 3 registers, "H" is for high and "L" is for Low part.

Since registers are located inside the CPU, they are much faster than a memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special

purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

Segment Registers

cs - points at the segment containing the current program.

ds - generally points at the segment where variables are defined.

ES - extra segment register, it's up to a coder to define its usage.

ss - points at the segment containing the stack.

Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose. pointing at accessible blocks of memory. This will be discussed further in upcoming classes.

Special Purpose Registers

- IP - The Instruction pointer. Points to the next location of Instruction in the memory.
- Flags Register - Determines the current state of the microprocessor modified automatically by the

CPU after some mathematical operations, determines certain types of results and determines how to transfer control of a program.

Writing Your First Assembly Code

In order to write programs in assembly language, you will need to familiarize yourself with most, if not all, of the instructions in the 8086 - instruction set. This class will introduce two instructions and will serve as the basis for your first assembly program.

The following table shows the instruction name, the syntax of its use, and its description. The operands heading refers to the type of operands that can be used with the instruction along with their proper order.

- REG : Any valid register
- memory : Referring to a memory location in RAM.
- Immediate : Using direct values.

Instruction	operands	Description
MOV	REG, memory memory, REG, REG, REG, memory, immediate REG, immediate	copy operand2 to operand1 • The mov instruction cannot : set the value of the cs and IP registers. • copy value of one segment register to another segment register (should copy to general register first). • copy an immediate value to segment register (should copy to general register first). Algorithm operand1 = operand2
ADD	REG, memory. memory, REG, REG, REG memory ; immediate REG, immediate	Adds two numbers Algorithm: operand1 = operand1 + operand2

bles that are not initialized.

Creating Constants

constants are like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants EQU directive is used.

name EQU <any expression>

For example:

```
K EQU 5
MOV AX, K
```

Creating Arrays

Arrays can be seen as chaines of variables. A text string is an example of a byte arry each caracter is a presented as an ASCII code value (0-225)

Here are some arry definition examples:

a DB 48h, 65h, 6Ch, 6Ch, 6fh, 00h

```
b DB 'Hello', 0
```

- You can access the value of any elemet in array using square brackets, for example:

```
MOV AL, a[3]
```

- You can also use any of the memory index registers BX, SI, BP, for example:

```
MOV SI, 3
MOV AL, a[SI]
```

- If you need to declear a large array you can use DUP operator

The Syntax for DUP:

number DUP (value(s))

number - number of duplicates to make (any constant value).

value - expression that DUP will duplicate

for example:

```
c DB 5 DUP (9)
```

is an alternative way of declaring:

```
c DB 9, 9, 9, 9, 9
```

one more example:

```
d DB 5 DUP (1, 2)
```

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Memory Access

To access memory, we can use these four registers: BX, SI, DI, BP. Combining these registers inside [] symbols, we can get different memory locations.

- Displacement can be an immediate value or offset of a valuable, or even both. If there are several values, assembler evaluates all values and calculates a single immediate value.

• Displacement is a signed value, so it can be bothe possetive and nagetive

Decleaning Array :

```
Array name db size DUP (?)
```

Value initialize :

```
arr1  db 50 dup (5,10,12)
```

Index Values :

```
mov bx, offset arr0
mov [bx], 6 ; inc bx
mov [bx+1], 10
mov [bx+9], 9
```

OFFSET :

"Offset" is assembler directive in x86 assembly language. It accually means "addres" and is a way of the heandling the overloading of the "mov" instruction.

Allow me to illustrate the usage -

1. mov si, offset variable

2. mov si, variable

The first line loads SI with the address of variable. The second line loads SI with the value stored at the adress of varriable.

As a matter of style, when I wrote x86 assembler I would write it this way -

1. mov si, offset variable

2. mov si, [variable]

The square brakets aren't necessary, but they made it much cleaer while loading the contents rather then the address.

LEA is an instruction that load "offset" variable" while adjusting the address between 16 and 32 bites as necessary. "LEA (16-bit register), (32-bit address)" loads the lower 16 bites of the address into the register, and "LEA (32-bite register), (16-bit address)" loades the 32-bit regis-

Topics to be covered in this class:

- Creating Variables
- Creating Arrays
- Creating Constants
- Introduction to INC, DEC, LEA instruct
- Learn how to access memory.

Creating Variable:

Syntax for a variable declaration:

name DB value

name DW value

DB - stands for Define Byte

DW - stands for Define Word

- name - can be any letter of dilite combination, though it should start with a letter. It's possible to declear unnamed variables by not specifing the name (this variable will have an address but no name)

- value - can be any numeric value in any supported numering system (hexadecimal, binary or decimal), or "?" symbol for varia-

Slide#3

Now form these on is compulsory i.e. Code Segment if at all you don't need variable(s) for you program, if you need variable(s) for you program you will need two Segments i.e. Code Segment and Data Segment.

First Line - DATA SEGMENT

DATA SEGMENT is the starting point of Data Segment in a program and DATA is the name given to this Segmet and Segment is the keyword of the difining segments, where we can declear our variables

NEXT Line – MESSAGE DB "HELLO WORLD!!!$"

MESSAGE is the variable name given to a Data Type(size) that is DB. DB stands for define Bites and in of One bite (8 bits). In Assembly language programs, variabls are defined by Data Size not its type. Chracter need One-

Byte so to store. Charactor and storing we need DB only that don't mean DB can't hold number or numerical value. The string is give in double quotes. $ is used as NULL charactor in C program so that compiler can understand where to STOP.

Next Line - DATA ENDS

DATA ENDS is the End point of the DATA SEGMENT in a Program. We cand write just ENDS But to differentiate the end of which segments it is of which we have to write the same name given to the Data segment.

Next Line - CODE SEGMENT

CODE SEGMENT is the starting point of the code segent in a program and CODE is the name to his segment and SEGMENT is the keyword for defining segment where

we can write the codeing of the program.

Next Line - ASSUME DS : DATA CS : CODE

In this Assembly Language Programing, there are Different Registers Present for Different Purpose So we have to assume DATA is the name given to Data Segment register and CODE is the name given to code segemnt register (SS, ES are used in the same way as CS, DS)

Next Line - START :

START is the label used to show the starting point of the code which is written in the Code Segment : is used to define a label as in C programming.

In this Assembly Language Programming, A single program is devided into four segments which are —

1. Data Segment
2. Code Segment
3. Stack Segment
4. Extra Segment

Print : Hello World in Assembly Language

```
DATA SEGMENT
     MESSAGE DB "HELLO WORLD!!!$"
ENDS
CODE SEGMENT
     ASSUME DS:DATA CS:CODE
START:
     MOV AX, DATA
     MOV DS, AX
     LEA DX, MESSAGE
     MOV AH, 9
     INT 21H
     MOV AH, 4CH
```