Laboratory Manual for
Computer Organization and Assembly Language

Instructor Abdul Qadeer Bilal

Semester

Department of Computer Science
National University of Computer and Engineering Sciences
Chiniot-Faisalabad Campus

Outline

- Structure of an Assembly Language Program
- Overview of Segmentation
- 8086 Registers and Their Functions
- Basic Instruction Set
- Example

Structure of an Assembly Language Code

An assembly language program is written according to the following structure and includes the following assembler directives. Each of the segments is called a logical segment. Depending upon the memory, the code and the data segments may be in the same or different physical segments.

TITLE
: Optional: Write the title of your program

.MODEL SMALL
: Assembler directive that defines the memory model to use in the program. The memory model determines the size of the code, stack and data segments of the program

.STACK
: Assembler directive that reserves a memory space for program instructions in the stack.

.DATA
: Assembler directive that reserves a memory space for constants and variables.

.CODE
: Assembler directive that defines the program instructions.

END
: Assembler directive that finishes the assembler program.

.MODEL Directive

Memory Model	Size of Code and Data
TINY	Code and data no more than 64KB combined
SMALL	Code and data segments must be no more than 64KB each
MEDIUM	Code can be more than 64KB, data still limited to no more than 64KB
COMPACT	Code limited to no more than 64KB, data can be more than 64KB
LARGE	Code and data can each be more than 64KB, no array can be larger than 64KB
HUGE	Code and data can each be more than 64KB, arrays can be larger than 64KB

Table 1: Memory Models

Overview of Segmentation

Segment means a large contagious chunk of memory in RAM which is used for storing data or program.

CODE Segment

It is also a large continuous chunk of memory in RAM which is used to store code.

1. Directive is **.code** for code segment.
2. The "program" resides here.

DATA Segment

It is also a large continuous chunk of memory in RAM which is used for storing variables.

1. Directive is **.data** for data segment.
2. All variables must be declared, and memory space for each allocated.
3. Data definition directive can be followed by a single value, or a list of values separated by commas.
4. Different data definition directives for different size types of memory.
    - **DB** - Define Byte (8 bits)
    - **DW** - Define Word (16 bits)
    - **DD** - Define Double Word (32 bits)
    - **DQ** - Define Quad Word (64 bits)

STACK Segment

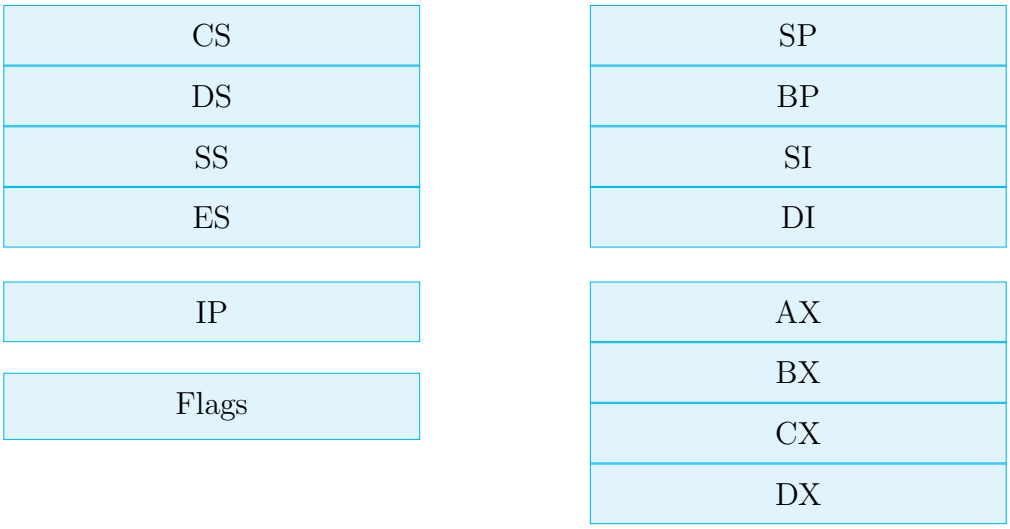
It is also a large continuous chunk of memory in RAM which is used for storing data or program.

1. Directive is .stack for stack segment.
2. Should be declared even if program itself doesn't use stack needed for subroutine calling (return address) and possibly passing parameters.
3. May be needed to temporarily save registers or variable content.
4. Will be needed for interrupt handling while program is running

8086 Registers and Their Functions

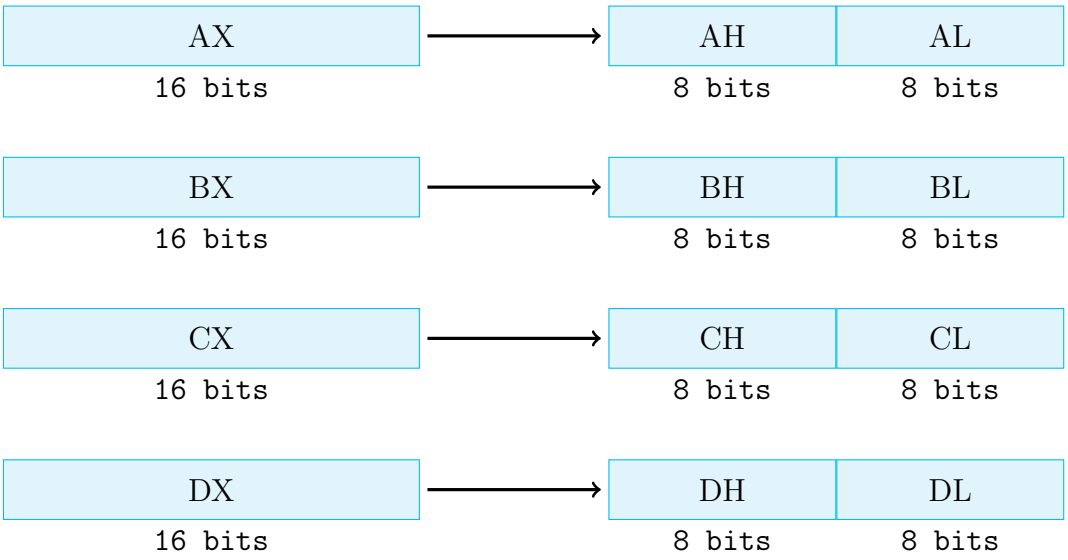
Registers are high-speed storage locations directly inside the CPU, designed to be accessed at mush higher speed than conventional memory.

The 8086 have 14 registers as shown below. Each register has some specific function associated with it, while writing the assembly language program you should have very strong knowledge about registers functionalities. If you don't know the register function properly you cannot write assembly program.



General Registers (AX, BX, CX and DX)

The registers AX, BX, CX and DX behave as general purpose registers and have some specific functions in addition to it. They are all 16-bit registers. For example AX means we are referring to 16-bit "A" register. Its upper and lower bytes are separately accessible as AH (A High Byte) and AL (A Low Byte). All general purpose registers can be accessed as one 16-bit register or as two 8-bit registers. The two registers AH and AL are part of the big whole AX. A change in AH or AL is reflected in AX as well.



These general registers in 32-bit processor are available in their extended form i.e. EAX, EBX, ECX and EDX where E pre-fix stands for extended or 32-bit. However in our emulator i.e. emu8086 we will mostly dealing with general register AX, BX, CX and DX not their extended form.

Specific Function of Each Register

AX
The A of AX stands for Accumulator. Traditionally all mathematical and logical operations are performed on the accumulator. Even though all general purpose registers can act as accumulator in most instructions there are some specific variations which can only work on AX (e,g. in multiplication and division instructions) which is why it is named the accumulator.

BX
The B of BX stands for Base because its role in memory addressing i.e. it can store memory address and then we can access that memory using this register.

CX
The C of CX stands for Counter as there are certain instructions that work with an automatic count in the CX register.

DX
The D of DX stands for Destination as it acts as the destination in I/O operations.

Index Registers (SI and DI)

SI and DI stand for source index and destination index respectively. These index registers of 8086 are used in memory accessing. These register can be used like general purpose register as there are many mathematical and logical operations that can be performed on them. However SI and DI are 16 bit registers and cannot be used as 8 bit registers like AX, BX, CX and DX. The source and destination are named because of their functionality in

string instructions where SI stores the address of source string and DI stores the address of destination string.

Instruction Pointer (IP)

This is the special register containing the address of the next instruction to be executed. No mathematical or logical operation can be performed on this register. It is out of our direct control and is automatically used. Instructions used in the program automatically change value of IP register. It is also of 16-bit size.

Stack Pointer (SP)

It is a memory pointer and is used indirectly in stack operations. Its size is 16-bit.

Base Registers (BP)

It is also a memory pointer containing the address in a special area of memory called the stack. Its size is 16-bit.

Flag Register

Flag register is not meaningful as a unit rather it is-bit wise significant and accordingly its each bit is named separately. We will discuss this register in detail later.

Basic Instructions Set

MOV

MOV instruction is used to transfer data between registers, between a register and a memory location, or to move a number directly into a register or a memory location.

Instruction	Explanation	Syntax	Example
MOV	Dest ← Operand or data stored at operand address	`MOV Dest, Src`	`MOV AX, BX`

Table 2: MOV Instruction

	Destination Operand			
Source Operand	**General Register**	**Segment Register**	**Memory Location**	**Constant**
General Register	Yes `MOV AX, BX`	Yes `MOV DS, AX`	Yes `MOV [100], AX`	No `MOV 5, AX`
Segment Register	Yes `MOV AX, CS`	No `MOV DS, CS`	Yes `MOV [100], CS`	No `MOV 5, CS`
Memory Register	Yes `MOV AX, SI`	Yes `MOV DS, SI`	No `MOV [100], [101]`	No `MOV 5, [100]`
Constant	Yes `MOV AX, 5`	No `MOV DS, 5`	Yes `MOV [100], 5`	No `MOV 5, 5`

Table 3: Logical Combination of Operands for MOV Instruction

ADD and SUB

The ADD and SUB instructions are used to add or subtract the contents of two registers, a register and a memory location, or to add (or subtract) a number to (or from) a register or memory location.

Instruction	Explanation	Syntax	Example
ADD	Dest ← Dest + Operand	`ADD Dest, Src`	`ADD AX, BX`
SUB	Dest ← Dest − Operand	`SUB Dest, Src`	`SUB AX, BX`

Table 4: ADD and SUB Instruction

	Destination Operand	
Source Operand	**General Register**	**Memory Location**
General Register	Yes `ADD AX, BX`	Yes `ADD [100], AX`
Memory Register	Yes `ADD AX, SI`	No `ADD [100], [110]`
Constant	Yes `ADD AX, 5`	Yes `ADD [100], 5`

Table 5: Logical Combinations of Operands for ADD and SUB Instruction

XCHG

This instruction swaps the contents of two operands, like in the above example data of AX and BX is being swapped.

Instruction	Explanation	Syntax	Example
XCHG	Operand 1 ↔ Operand 2 or data stored at operands address	`XCHG Dest, Src`	`XCHG AX, BX`

Table 6: XCHG Instruction

	Destination Operand	
Source Operand	**General Register**	**Memory Location**
General Register	Yes `XCHG AX, BX`	Yes `XCHG [100], AX`
Memory Register	Yes `XCHG AX, SI`	No `XCHG [100], SI`

Table 7: Logical Combination of Operands for XCHG Instruction

Complete Model for Assembly Program

```
01 .MODEL SMALL
02
03 .STACK 100H      ; Defines the size of stack segment
04
05 .DATA            ; Data Segment starts from here
06
07 .CODE            ; Code Segment Instruction set goes here
08
09
10
11 MOV AH, 4CH      ; To legally end the program
12 INT 21H
```

A Simple Program

Convert the following instructions to Assembly Language Instructions.

Instruction	Assembly Language Instruction	Register Value
Move 5 to AX	`MOV AX, 5`	AX = 05
Move 10 to BX	`MOV BX, 10`	BX = 0A
Add BX to AX	`ADD AX, BX`	AX = 0F
Move 15 to Bx	`MOV BX, 15`	BX = 0F
Add BX to Ax	`ADD AX, BX`	AX = 1E

Table 8: Simple Assembly Language Program

Example

To check the effect of basic instruction set on different registers.

```
01 .MODEL SMALL
02 .STACK 100H
03 .DATA
04 .CODE
05
06 MOV AX, 2000H
07 MOV BX, 2000
08 MOV CX, -2000H
09 XCHG BL, CH
10 MOV DH, 'A'
11 MOV AX, 'f'
12
13 MOV AH, 4CH
14 INT 21H
```

	Instructions to be executed					
	`MOV AX, 2000H`	`MOV BX, 2000`	`MOV CX, -2000H`	`XCHG BL, CH`	`MOV DH, 'A'`	`MOV AX, 'f'`
AX	**2000**	2000	2000	2000	2000	**0066**
BX	0000	**07D0**	07D0	**07E0**	07E0	07E0
CX	0015	0015	**E000**	**D000**	D000	D000
DX	0000	0000	0000	0000	**4100**	4100
IP	**0103**	**0106**	**0109**	**010B**	**010D**	**0110**

Table 9: Register values after executing the instructions