Assembly Language Programming

Programming with Intel 8086 microprocessor

Assembly Language Instruction Format:

Assembly language: Op-codes, mnemonics and operands

- Represented by certain words representing the operation of instruction. Thus programming gets easier.
- Generally written in a standard form that has four fields.
  - Label field
  - Op-code field (instruction or mnemonic)
  - Operand field
  - Comment field

Example:

Label	Mnemonic	Operand	Comment
Start:	MVI	A,10H	; MoveS 10H in register A
	MVI ADD HLT	B,04H B	; Move 04H in register B ; Add the content of register B with that of A ; Stop program

Mnemonic:

- short alphabetic code used in assembly language for microprocessor operation.
- words (usually two-to-four letter) used to represent each instruction.

Assembler:

- software (program module) which converts assembly language code (source module) into a machine language code.
- Assembly language program are called “source codes”.
- Machine language programs are known as ***object codes.***

Assembler types:

- **One Pass Assembler**:
  - Goes (scans) through the program once
  - Can resolve backward reference
  - Cannot resolve forward reference
- **Two Pass Assembler**:
  - Goes (scans) through the assembly language program twice.
  - First pass generates the table of the symbol, which consists of labels with addresses assigned to them.
  - Second pass uses the symbol table generated in the first pass to complete the object code for each instruction thus producing machine code.
  - It can resolve forward and backward reference both.

Backward reference	**Forward reference**
L1: …..	JMP L1
…..	…..
…..	…..
…..	…..
JMP L1	L1: …..

Linker:

- Program that links object file created by assembler into a single executable file.
- A translator converts source codes to object codes and then into executable formats.
- Converting source code into object code is called compilation and assembler does it.
- Converting object codes into executable formats is called linking and ***linker*** does it.

*.asm ~~assembler~~ *.obj ~~linker~~ *.exe

Macro assembler:

- an assembly language that allows macros to be defined and used.
- The Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows.
- translates a program written in macro language into the machine language.
- A macro language is the one in which all the instruction sequence can be defined in macro block.
- A macro is an instruction sequence that appears repeatedly in the program assigned with specific name.
- The MASM replaces a macro name with appropriate instruction sequence whenever it encounters a macro name.

- E.g.

```
Initiz macro
Mov ax, @dataseg
Mov ds, ax
Mov es, ax
Endm

 OR,
Initiz {
Mov ax, @dataseg
Mov ds, ax
Mov es, ax
}
```

- ***There exists little difference between macro program and subroutine program.***

- **Subroutine program:**
  - execution jumps out of main program and executes subroutine and control returns to the main program after RET instruction.

- **Macro:**
  - Does not cause program execution to branch out of main program.
  - Each time a macro occurs, it is replaced with appropriate sequence in the main program.

- **Advantages of using Macro:**
  - To simplify and reduce the repetitive coding.
  - To reduce errors caused by repetitive coding.
  - To make assembly language program more readable.

ASSEMBLER DIRECTIVES

- Instruction that will give the information to assembler for performing assembling are called assembler directives.
- Not executed by MP, so they are called dummy or pseudo instructions.
- It gives direction to the assembler.

8086 commonly used directives

- **Segment and Ends directive**
- **Proc and Endp directive**
- **END directive**
- **ASSUME directive**
- **PAGE directive**
- **TITLE directive**
- **EQU directive**
- **DUP directive Data Definition directive**
- **ORG directive**
- **Macro and Endm directive**
- **OFFSET directive**

Simplified Segment Directives

Memory Model	Description
Tiny	Code and data together may not be greater than 64K
Small	Neither code nor data may be greater than 64K
Medium	Only the code may be greater than 64K
Compact	Only the data may be greater than 64K
Large	Both code and data may be greater than 64K
Huge	All available memory may be used for code and data

- **. stack**
  - The .stack directive sets the size of the program stack, which may be any size up to 64K. This segment is addressed by SS and SP registers.

- **. code**
  - The .code directive identifies the part of the program that contains instructions. This segment is addressed by CS and IP registers.

- **. data**
  - All variables are defined in this segment area.

Addressing modes of 8086:

- There are 8 different addressing modes in 8086 programming:

1. ***Immediate addressing mode***

   The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.
   Example: MOV CX, 4929H, ADD AX, 2387H, MOV AL, FFH

2. ***Register addressing mode***

   The register is the source of an operand for an instruction.
   Example: MOV CX, AX; copies the contents of the 16-bit AX register into the 16-bit CX register),
   ADD BX, AX

3. Direct addressing mode

The addressing mode in which the effective address of the memory location is written directly in the instruction.

Example: MOV AX, [1592H],
MOV AL, [0300H],
ADD AX, [7765H]

4. Register indirect addressing mode

This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

Example: MOV AX, [BX]; Suppose the register BX contains 4895H, then the contents of 4895H are moved to AX

ADD CX, [SI]

5. ***Based (displacement) addressing mode***

In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

Example: MOV DX, [BX+04],
ADD CL, [BX+08]

6. ***Indexed addressing mode***

In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements.

Example: MOV BX, [SI+16],
ADD AL, [DI+16]

Instructions in 8086

- Instructions set found in 8086 are equivalent to 8085 but with the increase in operations variety, some few new instructions are added.
- Can be categorized as:
  - ✓Arithmetic,
  - ✓Data transfer,
  - ✓Logical and Bit manipulation,
  - ✓String,
  - ✓Program transfer and
  - ✓Process control.

1. Arithmetic Instructions

- **ADD (Addition):** Add specified byte to byte or specified word to word.
  - ADD R(8) /M(8) , R(8)/M(8)/ Immediate(8)
  - ADD R(16)/M(16), R(16)/ M(16)/ Immediate(16)
  - E.g. ADD AH, 15h; It adds contents of register AH and 15h immediately.
    - ADD AH, NUM1
    - ADD Al, [BX]
    - ADD [BX], CH/CX
    - ADD AX, [BX]

- **ADC:** Addition with Carry
  - **ADC** R/ M, R/M/Immediate data
- **SUB (Subtraction):**
  - SUB R/ M, R/M/Immediate data
- **SBB** (Subtraction with Borrow):
  - SBB R/ M, R/M/Immediate data
- **MUL**: unsigned multiplication
  - MUL R(8)/M(8) (8-bit Accumulator, AL)
  - MUL R(16)/ M(16) (16-bit Accumulator, Ax)
  - E.g. MUL R(8)(multiplier); AX (16-bit result)←R(8)×AL
    MUL R(16) (multiplier); DX:AX (32-bit result)←R(16)×AX

- **IMUL**: signed multiplication
  - *Same as MUL operation but takes sign into account.*
- **DIV (Division):**
  - DIV R(8); AX/R(8) Remainder in AH and Quotient in AL
  - DIV R(16); DX:AX/R(16) Remainder in DX and Quotient in AX
- **IDIV (Signed Division):**
  - Same operation as DIV but takes sign into account.
- **INC/DEC (Increment/Decrement by 1):**
  - Increment/decrement the Register content by unity.
  - INC/DEC R/M, (8-bit or 16-bit)
  - E.g.: INC AL, DEC BX, INC num1

- **NEG (Negate – 2's Complement):**

ASCII-BCD conversion:

- ***AAA (ASCII adjust after addition)***
  - Corrects result in AH and AL after addition when working with BCD values.
- ***DAA (decimal adjust after addition)***
  - Corrects the result of addition of two packed BCD values
- ***DAS (decimal adjust after subtraction)***
- ***AAS (ASCII adjust after subtraction)***
- ***AAM (adjust after multiplication)***
- ***AAD (adjust after division)***

Logical/shifting/comparison Instructions

- **Logical**
  - ✓AND/OR/XOR R/M, R/M/Immediate
  - ✓NOT R/M; Invert each bit of byte or word.
  - ✓E. g. AND AL, AH
  - ✓XOR [BX], CL

- **Rotation**
  - ROL/ROR/RCL/RCR**,** R/M, 1/CL
  - *ROL- rotate left,*
  - *ROR-rotate right*
  - E.g. ROL AX, 1; rotated by 1
  - ROL AX, CL; if we need to rotate more than one bit
  - *RCL-rotate left through carry*
  - *RCR-rotate right through carry*
  - E.g. RCL AX, 1
  - RCL [BX], CL; Only CL can be used.

- **Shifting**
  - SHL/SHR/SAL/SAR, R/M,1/CL
  - ***SHL - logical shift left***
  - ***SHR - logical shift right***
  - *Shifts bit in true direction and fills zero in vacant place.*
  - ***SAL - arithmetic shift left***
  - ***SAR - arithmetic shift right***
  - *Shifts bit/word in true direction, in former case place zero in vacant place and in later case place previous sign in vacant place.*
  - E.g. SHL AX, 1; rotated by 1
  - SHL AX, CL; if we need to rotate more than one bit
  - SAR DX, 1
  - SAR [BX], CL; Only CL can be used.

- **Comparison**
  - CMP –compare
  - CMP R/M, R/M/Immediate
  - E.g. CMP BH, AL

Operand1		Operand 2	CF	SF	ZF
	>		0	0	0
	=		0	0	1
	<		1	1	0

Data Transfer Instructions

- Move bytes or words of data between memory and registers as well as between the registers and the I/O ports.
- **Mov** r/m, r/m/immediate
  - Copy byte or word from specified source to specified destination.
- **In** al/ax, dx
  - Copy a byte or word from specified port number to accumulator.
  - Second operand (dx) is a port number. If required to access port number over 255 - **DX** register should be used.
  - E.G.: In al, dx
    Out dx, al/ah

- **OUT** DX, AL/AX
  - Copy a byte or word from accumulator to specified port number, first operand (DX) is a port number. If required to access port number over 255 - **DX** register should be used.
- **LEA** R, M
  - Load effective address of operand into specified register
- **LDS** BX, NUM1 (Load data segment register)
- **LES (**Load extra segment register)
- **LSS** (load stack segment register)

- **XCHG** R/M, R/M/immediate
  - E.g.: XCHG AX, BX
    XCHG AL, BL
    XCHG CL, [BX]

- **Flag Operation:**
  - CLC: Clear carry flag
  - CLD: Clear direction flag
  - CLI: Clear interrupt flag
  - STC: Set Carry flag
  - STD: Set direction flag
  - STI: Set Interrupt flag
  - CMC: Complement Carry flag
  - LAHF: Load AH from flags (lower byte)
  - SAHF: Store AH to flags
  - PUSHF: Push flags into stack
  - POPF: Pop flags off stack

- **STACK Operations:**
  - PUSH R (16-bit)
  - POP R (16-bit)
- **Looping instruction:** Register CX is automatically used as a counter.
  - LOOP: loop until complete
  - LOOPE: Loop while equal
  - LOOPZ: loop while zero
  - LOOPNE: loop while not equal
  - LOOPNZ: loop while not zero

- **Conditional**
  - JA (Jump if above)
  - JAE (Jump if above/equal)
  - JB (Jump if Below)
  - JBE (Jump if below/equal)
  - JC (Jump if carry)
  - JNC (Jump if no carry)
  - JE (Jump if equal)
  - JNE (Jump if not equal)
  - JZ (Jump if zero)
  - JNZ (Jump if no zero)
  - JG (Jump if greater)
  - JNG (Jump if no greater)
  - JL (Jump if less)
  - JNL (Jump if no less)
  - JO (Jump if overflow)
  - JS (Jump if sign)
  - JNS (Jump if no sign)
  - JP (Jump if plus)
  - JPE (Jump if even parity)
  - JNP (Jump if no parity)
  - JPO (Jump if parity odd)

Unconditional

- CALL (Call a procedure)
- INT (Interrupt)
- JMP (Unconditional jump)
- RETN/RETF (Return near/far)
- RET (Return)
- IRET (Interrupt return)

- **Type conversion:**
  - **CBW (Convert byte to word)**
  - **CWD (Convert word to double word)**
- **String instructions:**
  - MOVS/MOVSB/MOVSW; Move string
  - *DS: SI (Source)*
  - *DS: DI (Destination)*
  - *CX (String Length)*
  - CMPS/CMPSB/CMPW; Compare string
  - LODS/LODSB/LODW;Load string
  - REP; Repeat string

INT 21h functions

- Provided in the photocopy
- DOS services
- 01h, 02h, 09h, 0Ah, 4Ch

INT 10h functions

- Provided in the photocopy
- Video display services (BIOS services)
- 00h, 01h, 02h, 06h, 07h, 08h, 09h, 0Ah

Function number	**Description**
01h e.g. mov ah,01h int 21h	Keyboard input with echo: This operation accepts a character from the keyboard buffer. If none is present, waits for keyboard entry. It returns the character in AL.
02h e.g. mov ah,02h int 21h	Display character: Send the character in DL to the standard output device console.
09h e.g. mov ah, 09h Int 21h	String output: Send a string of characters to the standard output. DX contains the offset address of string. The string must be terminated with a „$“ sign.
0Ah	String input
4Ch e.g. mov ax,4C00h int 21h	Terminate the current program.

Function number	**Description**
00h	Set video mode
01h	Set cursor size
02h	Set cursor position
06h	Scroll window up
07h	Scroll window down
08h	Read character and attribute of cursor
09h	Display character and attribute at cursor
0Ah	Display character at cursor