ASSEMBLY LANGUAGE

INSTRUCTION SET

This is a set of predefined operation called OP-CODES that a microprocessor is designed to recognize and perform. Each type of microprocessor has a unique instruction set. A typical assembly language machine instruction consists of 4 fields that is Label, Mnemonics (OP-*CODE),* Operand field and Comment field. The assembler ignore comment field but translates the other 3 fields.

#1	#2	#3	#4
LABEL	MNEMONIC	OPERAND(S)	COMMENT

LABEL: Serves as identification for the statements it precedes. It is not mandatory to precede instruction in assembly language by line number. Any instruction that requires identification is proceeded by identification name known as label.

A label is made up of alphanumeric characters not more than 6, the first of which must be a letter. During assembly, the assembler it replaces the name by the memory address of the first bytes of the instruction object code it precedes.

OPERAND(S): This field contains the operands that will be operated on by the preceding OP-CODE. Operations may require one or two operands, MONADIC requires one operand while DIADIC requires two operands.

COMMENT: This field contains clarifying information on instruction or a group of instructions. It is information to the user, it neither generated object code nor serve as directive to the assembly. Only field #2 is always mandatory to be present in any machine instruction except in a line entirely devoted to comment.

Label is separated from mnemonics by (**:**) colon immediately after the label name, otherwise the field is left blank. The Mnemonics field is separated from the operand field by at least a space (blank). If operand field contain more than one operand are separated by comma from one another, but if the operand is separated from comment by a semicolon (**;**) symbol.

3.1 INSTRUCTION FORMAT

Any instruction or data word must convey the information to operation to be performed(Op-code) of the address of the memory locations or register containing the

data (operand) on which operation is to be performed. Instruction in a CPU are of different kinds based on the number of operand they contain or require to operate on. These are as follows:

- Zero Address Format: Instruction that do not require any address or operand. The operand address is implied. Examples: CLA (which clear the accumulator). STC (set carry), NOP etc.

One Address Format: A single operand address is specified. The operand lies in the accumulator and the result is also stored back in the accumulator.

<OP-CODE> Addr 1

Examples: ADD C; ACC ← ACC + C

LDA B; ACC ← B

MUL D; ACC ← ACC* D

- Two Address Format: Address of two Operands are specified. The result of the operation is stored in one of the given operand address. <OP-CODE> Addr 1, Addr 2

Examples: MOV A, R_1: $R_1 \leftarrow A$ Addr1= source

SUB R_1, R_2: $R_2 \leftarrow R_2 - R_1$ Addr2 = destination

ADD C, R_2: $R_2 \leftarrow R_2 + C$

- Three Address Format: Two operand address are specified and the third address is provided for storing the result. <OP-CODE> Addr 1, Addr2, Addr 3

Examples: MUL A, B, C: C ← A*B

ADD A, B, C: C ← A+B

SUB R_1, R_2, R_3: $R_3 \leftarrow R_1 - R_2$

3.2 8086/8088 INSTRUCTION SET

The instruction set of 8086 and 8088 have exactly the same instruction set. There are about 100 functional or generic instructions. The purpose of instruction set is to facilitate development of efficient program users, it also reflects the power of the underlying architecture that users can take advantage of while developing programs. Programming in 8086/8088 assembly language only involves selecting appropriate mnemonics from

the instruction set table with the possible addressing modes. Instructions can classified into the following groups:

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- String Instructions
- Transfer of Control Instructions
- Process Control Instructions

1. *Data Transfer Instruction:* There are 14 data transfer instructions that are responsible for moving single bytes or words between memory and registers, between registers and between register AL or AX and I/O ports. Included in these categories are the stack manipulation, instructions for transferring flag contents and instructions for loading segment registers. They are:

MOV	XLAT	LDS	PUSHF
PUSH	IN	LES	POPF
POP	OUT	LAHF	
XCHG	LEA	SAHF	

General Purpose Transfer Instruction:

MOV dst, src; move a byte or word operand from source to destination dst = segment register, register ;or memory operand src = memory, register, segment register.

Example:MOV AX, [BX], MOV array [s_i], a1.

PUSH sr; Increment SP (stack pointer) register by 2, move then a word operand from source to the stack. src = register, memory operand, segment register.

Examples: PUSH DX, PUSH ES, PUSH return_code [SI]

POP dest; Transfer a word operand from the current top of the stack, TOS, to destination, then decrement SP by 2. SP always points to the TOS dst = register, segment register, memory operand.

Examples: POP DX, POP, mem_label POP DS.

XCHG dst, scr; Exchange a byte or word operand content of the source with that of the destination dst = register, memory, src = memory, register.

Examples: XCHG semaphore ax, XCHG dl ch, XCHG ax, bx.

XLAT src_tablel; Use the contents of AL register as an index into a table whose base address is pointed to by register BX. It has the effect of translating from one code to another code.

Examples: MOV al 30, LEA BX, EBCDIC_table, XLAT EBCDIC_table [AL]

IN dst, src ; Transfer a byte or word from an input port source to destination accumulator dst = AL for btyes transfer or AX for word transfer, src number.

Examples: IN al OFFEAH, IN ax, dx.

OUT dst, scr; Transfers a byte or word from register to output port address. dst = out_port_number. Src = for byte transfer or AX for word transfer.

Examples: OUT 44, ax , OUT dx, al

Address Object Transfer Instructions

These instructions take the address of a variable rather than its contents or value.

LEA dst, src; Load Effect Address, transfer the offset address of a source operand to destination operand.

Examples: LEA BX, EBDCDIC_table, LEA BX, [BP+DI].

LDS dst, src; Load pointer using DS, transfers a 32 bits pointer variable from the source to destination operand and segment register DS.dst = DS: 16 bit general register src = 32 bit memory operand.

LDS S1, TABLE; where SI = offset address of TABLE, DS = higher_word of TABLE

LDS dest, src; Load pointer using ES

Example:LES D1, TABLE2

Flag Transfer Instructions:

LAHF; Load register AH from the processors flag, it copies SF, ZF, AF, PF and CF flag into bits 7, 6, 4, 2 and 0 respectively to register AH.

SAHF; Store register AH into processor flag, it transfers bit 7, 6, 4 2 and 0 from register AH to SF, ZF, AF, PF and CF respectively. DF, TF and IF are not affected.

PUSHF; Pushes the 16 bit contents of all the processor's flag onto stack.

POPF; Transfer 16 bit word from TOS to the processor's flag register.

2. Arithmetic Instruction: These are performed on 4 data types: unsigned binary, signed binary (integer), unsigned packed decimal and unsigned unpacked decimal. These instruction are: ADD SUBB MUL CWB

ADC DEC IMUL CWD

INC NEG AAM AAA CMP DIV

DAA AAS IDIV SUB DAS AAD

ADD dest, src; Add a byte or word source operand to a destination operand (dst = dst + src).

Examples: ADD ax, bx , ADD cl, 2

ADC dst, src; Add with carry, same as ADD but the content of carrying is added.

Examples: ADC ax, s_i ; ADC bx, 256

INC dst; Increment a byte or word operand by 1, dst = dst + 1

Examples: INC bx ; INC [bx +di + alpha]

AAA; ASCII Adjust for Addition. It changes the contents of register AL to a valid unpacked decimal number.

Examples: Add al , bl. If AL = 32h, BL= 33h

AAA AL = 35h

DAA; Decimal Adjust for Addition corrects the result of two valid packed operands previously added.

Examples: Add al, bl. If al = 46h, bl = 27h

DAA then al = 73h

SUB dst , src; Subtract a byte or word source operand from hat of destination operand. dst = dst – src

Examples: SUB ax, bx ; SUB d_i , [bx + alpha]

SBB dst, src; Subtract with borrow same as SUB but the content of the carry flag is also subtracted.

Examples: SBB ax , s_i ; SBB bx , 256

DEC dst; Decrement a byte or word operand by 1

Examples: DEC bx ; DEC [bx + d_i + alpha]

NEG dst; Negate a byte or word operand performs the 1’s complement.

Examples: NEG dl ; NEC mem_var

CMP dst , src; Compare a byte or word source operand with that of the destination.

Examples: CMP ch , 02 ; CMP bl , [s_i]

AAS; ASCII Adjust for Subtraction. Same as AAA except subtraction takes place.

DAS; Decimal Adjust for Subtraction.

MUL src; Multiply an unsigned byte or word. It multiplies unsigned source operand and the Accumulator.

Examples: MUL cl → ah: al = al* cl

MUL bx → dx: ax = ax * bx

IMUL src; Integer multiplication. If integer multiplies a byte or word signed operand and accumulator.

Examples: IMUL cl ; IMUL rate_byte.

AAM; ASCII Adjust for Multiply. It corrects the result of a previous multiplication.

DIV src ; Divide perform unsigned division of the accumulator by the source operand.

Examples: DIV cl ; DIV [s_i + table].

IDIV src ; Perform signed binary division

Examples: IDIV bl ; IDIV [s_i + divisor_byte]

AAD; ASCII adjust for Division

CWB; Convert byte to word

CWD; Convert word to double word

3. Bit Manipulation: These include Logical Instruction, Shift and Rotate.

Logical Instruction: The instruction sets of typical microprocessor include instruction in order to perform the Boolean AND, OR, NOT and an EXCLUSIVE-OR operation or bit-by- bit.

NOT dst; Perform that 1's complement of a byte or word destination operand.

Examples: NOT ax ; NOT table

AND dst , src; Perform logical AND of a byte or word operand of source with that of destination dst = dst AND src

Examples: AND al , bl ; AND cx , flag_word.

OR dst, src; Performs logical inclusive OR of a byte or word operand of source with that of destination dst = dst OR src.

Examples: al , bl ; OR cx , flag_word.

XOR dst , src; Performs logical exclusive OR of a byte or word operand of source with the destination dst = dst XOR src

Examples: XOR al , bl

TEST dst , src; Performs logical And of a byte or word operand of source with that of destination but does not return the result into the destination it affects only the flags instead. Set flags = dst AND src.

Examples: TEST si , di ; TEST al , 00100110B

Shift Instructions:

SHL dst, count and SAL dst, count; Both shift left logically (SHL) and shift left arithmetically (SAL) have the same function and are equivalent. The destination operand is shift left the number of times specified in COUNT, zero are shift in on the right, COUNT= 1 or reg CL if count is more than 1

Examples: SHL al , 1 ; SAL di , cl.

SHR dst , count; Shift right logically, zero are shifted in from the left.

Examples: SHR si , 1 ; SHR si, cl

SAR dst , count; Shift right arithmetic same as SHR except that the sign bit i.e. MSB is shifted in instead of zero, from the left.

Examples: SAR si , cl ; SAR [bx +si + id_byte] , cl

Rotate Instruction:

ROL dst, count; Rotate left into CF a byte or word destination operation by the number of times specified in COUNT. It shift the leftmost bit into CF and into the rightmost bit position.

Examples: ROL bx , 1 ; ROL [di +flag_byte] , cl

ROR dst , count; Rotate left into CF, same as ROl except it is the rightmost bit that is shifted in CF and the leftmost bit position.

Examples: ROR bx , 1 ; ROR [di +flag_byte] , cl

RCL dst , src: Rotate left through carry, it shifts flfom the left the dst operand including the CF the number of times specified by count. The value of CF is rotated into LSB of dst.

Examples: RCL bx 1 ; RCL [di +flag_byte] , cl

RCR dst , count; Rotate right through carry, same as RCL except the rotation is to the right.

4. String Manipulation: For string manipulation, the hardware assumes that a source string is in the current DS (source string) while destination string must be in ES (destination offset/string).

String Instruction Table

Mnemonics	**Description**
	Repeat for a specified number of iteration value in register CX.
REPE, REPZ	The two are the same, Repeat while the ZF=0. The maximum number of iteration should be specified in register CX
REPNE REPNZ	Both are the same, Repeat while ZF≠ 0, the maximum number of iterations should be specified in register CX.
MOVS	Moves a string a bytes/ words from location to another in memory.
SCAS	Scan string for specified values
CMPS	Compare values in source string with values in destination string.
LODS	Load values from a string to accumulator register.
STOS	Store values from accumulator register to a string.

String Instruction Register and Use

Register	**Use**
SI	Source String Offset
DI	Destination String Offset
CX	Repetition Counter
AL	Scan value register, source for STOS
AX	Destination for LODS
DF	DF=0 means auto increment of SI, DI DF=1 means auto decrement of SI. DI
ZF	Scan and compare terminator

A string instruction syntax is one of the following:

repeatprefix STRING_MNEMONIC

repeatprefix string_mnemonicB (byte)

repeatprefix string_mnemonicW (byte)

ES: destination_offset

Seg_reg: source_offset.

NOTE:

1. The operand destination_offset and source_offset only indicate the size of the data object to be processed by the string. The actual values of operands are pointed to by ES:DI for destination string and DS: SI for the source string.The source operand segment can be overridden but that of destination, ES reg., cannot be changed
2. The letter B or W appended to the string instruction indicate a byte or word operand.
3. Sting instruction with implicit operand size such as B or W require no further operands. Those strings instruction without implicit operand size e.g. MOVS must include destination_offset or/and source_offset operand for the required size information.

SETTING UP STRING OPERATIONS

The general procedure for setting up string operation is as follows:

1. Set up the DF. Use CLD instruction to clear it or STD to set it.
2. Load register CX with the number of desired iterations.
3. Load the starting offset address of the source string into DS:SI and the starting offset address of the destination string into ES:DI.
4. Choose the appropriate REPEAT_PREFIX instruction.
5. Put the appropriate STRING_MNEMONIC after the prefix.

Repeat Prefix	Mnemonics & operand	Remarks
REP	MOVS dst, src	dst = byte/word, the memory operand ID.src = byte/word memory operand ID
REP	MOVSB MOVSW	Implicit operand ID, repeat until reg. CX = 0
REPE REPNE	SCAS dst	dst = byte/word memory operand reg. AL = scan_value, repeat if ZF = 1 and CX ≠ 0 for REPE, repeat if ZF = 0 and CX ≠0 for REPNE.
REPE REPNE	CMPS dst, src	dst = byte/word memory operand src = byte/word memory operand repeat if ZF = 1 &CX ≠0 for REPE, repeat if ZF = 0 & CX≠0 for REPNE .
REPE	CMPSB	Same as CMPS but for byte operand.
REPNE	CMPSW	Same as CMPS but for word operand

REP	LODS src	src = byte/word memory operand reg. AL = src_operand. Repeat until CX = 0.
REP	LODSB LODSW	Implicit operand ID same as LODS for byte Implicit operand same as LODS for word
REP	STOS dst	dst = byte/word memory operand ID. dst_string = reg AL, until CX =0
REP	STOS B STOSW	Implicit operand ID same as STOS for byte Implicit operands ID same as STOS for word.

NOTE:

For all string instructions

dst = string_pointer = ES: DI

src = string_pointer = DS : SI

If DF = 0, then DI = DI + 1 or DI = DI +2

For byte or word operand respectively after each string operand.

If DF = 1 then DI = DI – 1 or DI = DI – 2 for byte or word operand respectively after each string operation.

5. Program Control Instruction: The flow of control depends on the result of computation. A program can select a particular sequence of instruction to execute based on the results of computation. It can be classified as follows:

Unconditional Branch instruction: It transfers the control to the specified address regardless of the status of the computation.

Conditional Branch Instruction: If (Condition) then branch to execute a new instruction else execute the following instruction.

LDA BEGIN ; Load ‘A’ with [BEGIN]

MVI C, 03 ; C← 03

SUB C ; [A] ← [A] – [C]

JZ START ; Jump to START if Z= 1

MOV A, C ; If Z=) do this

.
.
.

Start HLT ; Half

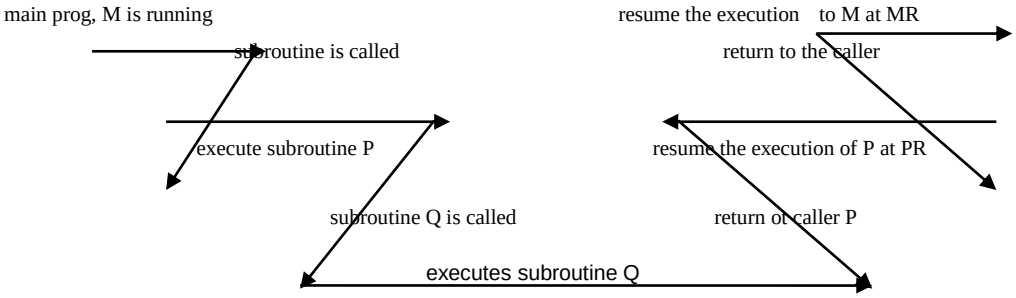
Subroutine Call Instruction: The Subroutine Calls and Returns from subroutine are usually handled by two special instruction namely:

Call Instruction: This is of the form CALL (addr) where the parameter (addr) refers to the address of the first instruction of the subroutine when the instruction is encountered the current content of the PC are saved in the stack and the PC is loaded with addr.

The current content of the PC provided the address of the instruction that immediately follows the CALL instructions, this address is also called RETURN ADDRESS because this is the point where execution of the calling program will take place after existing from the subroutine.

For example: Consider a main program M and two subroutine P and Q as in Fig 1, the main program calls subroutine P and this subroutine in turn calls subroutine Q the expected control flow sequence is as in Fig 2. The parameter MR and PR refers to the return addresses of the main program M and the subroutine P respectively. When the main program calls the subroutine P, the subroutine Q, the return address PR is pushed onto stack and the control is transferred to the subroutine Q. When the subroutine Q completes its execution, the return address is retrieved from the stack and loaded into the PC.

;Main Program		
……	P……	Q……
……	……	……
CALL P	CALL Q	……
MR……	PR……	……
	……	……
END	RET	RET
Main Program	Subroutine P	Subroutine Q



6. Input/Output (I/O) Instructions: The I/O operation is defined as the transfer of data between the microcomputer system and the external word. It is contain in the ROM and RAM chip. An input instruction allows peripheral to transfer a word to either a register or the main memory. The output instruction enables a processor to transfer a word into the buffer register of the peripheral.

There are typically three main ways of transferring data between the microcomputer system and external devices. They are:

- *Programmed (I/O):* This technique the microprocessor executes a program to perform all data transfers between the microcomputer system and the external devices via one or more registers called I/O ports.

Characteristics: The external devices carry out the function as dictated by the program inside the microcomputer memory

- *Interrupt I/O:* Microcomputer can transfer data to or from an external device using the interrupt I/O in order to accomplish this, the μcomputer uses a pin on the μP chip called the interrupt pin (INT), the external device is connected to this pin. When the device wants to communicate with the μcomputer, it makes the signal on the interrupt line HIGH or LOW depending on the microcomputer. In response the microcomputer completes the current instruction and pushes at least the contents of the current program counter and may be some other internal registers onto the stack. It then automatically loads an address into the program counter in order to branch to a subroutine like programme written by the user. This programme is called *interrupt service routine.*

This is a programme that the external device wants the microprocessor to execute in order to transfer data. The last instruction of the service routine is a RETURN instruction which is the same instruction typically used at the end of the subroutine. This instruction

pops the address (which was pushed unto the stack before going to the service routine) from the stack into the PC. The microcomputer then continues in the main programme that it had been executing.

- *Direct Memory Access (DMA):* Programmed Input/Output and interrupt Input/Output (I/O) provide data transfer between the μP and external device. However, there are various instances when data must be moved between memory and the external devices. E.g. with a mass storage device like flash drive, one may want to input or output programs or data to or from the computer RAM. DMA transfers data directly between the RAM and an Input/Output device without involving the μP. Using this technique, the transfer of 1 byte of data typically requires RAM but DMA is extensively used in transferring large blocks of data between a peripheral device and the microprocessor memory. An interface chip called the DMA controller clip is used with the UP for transferring data via DMA. Since DMA performs data transfer between memory and an extend device without involving the μP, the DMA interface or controller chip must be able to perform memory READ and WRITE operations in a similar way as the μP.

3.3 ADDRESSING AN I/O PORT

Generally, the movement of data from the μP between to the I/O device and vice verse is achieved in 2 ways

a. Standard I/O
b. Memory Mapped I/O

Standard I/O: Typically utilizes a control pin on the μP chip commonly called the IO/M control signal. A HIGH on this pin indicates an I/O operation, whereas a low indicate a memory operation. The execution of IN and OUT instruction is said to utilize standard I/O.

Memory-Mapped I/O: Here, the μP uses RAM addresses to represent I/O ports. The port devices are addressed and selected by decoders as if they were memory devices, with memory-mapped I/O, one uses 3- byte instructions namely LDA and STA as follow:

LDA
XX } I/O port address } 3-byte instruction for inputting
XX } mapped into memory } a byte into accumulator.
STA
XX } I/O port address } 3-byte instruction for outputting
XX } mapped into memory } data from accumulator into
specified I/O port.

Whereas standard I/O, one typically uses 2-byte instructions namely IN and OUT as follows:

IN port number } 2-byte instruction for inputting
XX } data from specified port accumulator.

OUT port number } 2-byte instruction for outputting
XX } data from accumulator into specified I/O port.

Advantages	Disadvantages
In memory mapped I/O, any instructions which refers to memory can theoretically be used to read from or write to a port.	The port data is moved into the accumulator.
Since the I/O port address are configured as memory addresses, one can use all the μPs instruction that reference memory addresses for the I/O ports. The means that one can perform arithmetic and logic operations and ant other functions on port data.	Standard I/O in order to arithmetic and logic operation, one must move the port data into the accumulator.
3-byte instruction such as LDA and STA are required for inputting and outputting a byte.	2-byte instructions such as IN and OUT are used for INPUTTING OR OUTPUTTING a byte. The I/O address space is smaller and hence requires a 1 byte address to specify the I/O address.
The ports occupy part of the system memory space. This space is then not available for storing data or instructions. Thereby reducing the maximum size of the memory.	Have a separate address space for ports and in accessed directly with the In and OUT instructions