Assembly Language

Introduction

- Machine Language Programming

✓One way to write a program is to assign a fixed binary pattern for each instruction and represent the program by sequencing these binary patterns.

✓Such a program is called a Machine language program or an object program

Eg:

```
0011    1110    ; LOAD  A  Register with
0000    0101    ; Value 5
0000    0110    ; LOAD  B  Register with
0000    0110    ; Value 10
1000    0000    ; A  <-  A  +  B
0011    1010    ; store the result
0110    0100    ; into the memory location
0000    0000    ; whose address is 100
0110    0110    ; halt processing
```

Assembly Language Programming

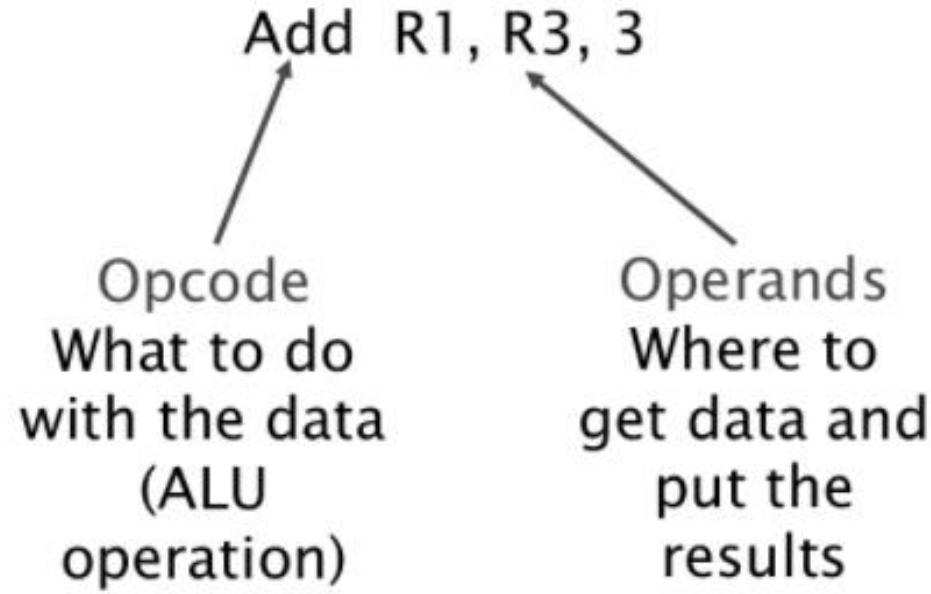
- ✓ Use of symbols in programming to improve the readability.
- ✓ Giving symbolic names for each instruction.
- ✓ These names are called **mnemonics** and a program written using these symbols are called as **Assembly Language Program**.

mnemonics

- In assembly language, mnemonics are used to specify an opcode that represents a complete and operational machine language instruction. This is later translated by the assembler to generate the object code. For example, the mnemonic MOV is used in assembly language for copying and moving data between registers and memory locations.

Assembly Language Instructions

- Built from two pieces



Eg:

```
LOAD   A, 5          ; load A reg with 5
LOAD    B, 10        ; load B reg with 10
ADD  A, B            ; A = A + B
LOAD  (100), A       ; save the result in
                       location 100
HALT                 ; halt processing
```

Format of Assembly program

LABEL	OPCODE	OPERAND	COMMENTS

1. **Labels** :Begins from column1. Should start with a letter. After the letter a combination of letters and numbers may be used. Restricted by most assemblers from 6 to 8 characters. Use is to identify opcodes and operands. Are needed on executable statements, so that the statement can be branched to.

- **2. Opcode**: Symbolic abbreviations for operation codes or comment to the Assembler. Specifies how data are to be manipulated, which is residing within a microprocessor register or in the main memory. Eg: MOV, LD, ADD
- **3. Operand**: use to specify the addresses or registers used by the operands by the Machine instructions. Operands may be Registers, memory location, constants etc.
- **4. Comments**: prefixed by ; for documentation

Eg.

LABEL	OPCODE	OPERAND	COMMENTS
ADD	AL,	078h	;ADD 07H to AL register

Features of ALP

- For using Assembly language, the programmer needs to know the internal Architecture of the Microprocessor.
- The Assembly language written for one processor will not usually run on other processors.
- An assembly language program cannot be executed by a machine directly, as it not in a binary form.

- An **Assembler** is needed in order to translate an assembly language (source pgm) into the object code executable by the machine.

- Unlike the other programming languages, assembly language is not a single language, but rather a group of languages.

- Each processor family (and sometimes individual processors within a processor family) has its own assembly language.

Comparison of Assembly language with High level languages

- Assembly languages are close to a one to one correspondence between symbolic instructions and executable machine codes.

- Assembly languages also include directives to the assembler, directives to the linker, directives for organizing data space, and macros.

- Assembly language is much harder to program than high level languages. The programmer must pay attention to far more detail and must have an intimate knowledge of the processor in use.

- High level languages are abstract. Typically a single high level instruction is translated into several (sometimes dozens or in rare cases even hundreds) executable machine language instructions.

- Modern object oriented programming languages are highly abstract

- But high quality assembly language programs can run much faster and use much less memory and other resources than a similar program written in a high level language.

- Speed increases of 2 to 20 times faster are fairly common, and increases of hundreds of times faster are occasionally possible.

- Assembly language programming also gives direct access to key machine features essential for implementing certain kinds of low level routines, such as an operating system kernel or microkernel, device drivers, and machine control.

- High level programming languages are much easier for less skilled programmers to work in and for semi-technical managers to supervise.

- High level languages allow faster development times than work in assembly language, even with highly skilled programmers.

- Development time increases of 10 to 100 times faster are fairly common.

- Programs written in high level languages (especially object oriented programming languages) are much easier and less expensive to maintain than similar programs written in assembly language (and for a successful software project, the vast majority of the work and expense is in maintenance, not initial development).

Advantages:-

- **Performance**: An expert Assembly language programmer can often produce code that is much smaller and much faster than a high level language programmer can. For some applications speed and size are critical. Eg:- code on a smart card, code in a cellular telephone, device drivers, BIOS routines etc.

2. **Access to the Machine**: Procedures can have complete access to the Hardware (Not possible in High level language)

eg: low level interrupt and trap handlers in OS

8086 Instruction Set

Can b classified as shown:

- 1. Data Transfer Instructions
- 2. Arithmetic Instructions
- 3. Bit Manipulation Instructions
- 4. String Instructions
- 5. Program Execution Transfer Instructions.
- 6. Processor Control Instructions.

1. Data-transfer Instructions

A. General purpose byte or word instructions

- **MOV**: copy byte or word from specified source to destn. MOV AX, BX
- **PUSH**: copy specified word to top of the stack. PUSH BX
- **POP**: copy word from TOS to specified location. POP CX
- **PUSHA/POPA**: copy all registers to stack and copy words from stack to all regs.
- **XCHG**: Exchange byte or word.
  XCHG DX, AX

B. Simple I/O port transfer instructions

- **IN** :copy a byte or word from specified byte to accumulator. IN AL, 028h
- **OUT** :copy a byte or word from accumulator to specified port. OUT 028h,AL

C. Special Address transfer Instructions

- **LEA**: Load Effective Address of operand to specified register. LEA BX, START.
- **LDS**: Load DS register and other specified register from memory.
- **LES**: Load ES register and other specified register from memory.

D.Flag transfer instructions

- **LAHF**: Load (Copy to) AH with the low byte of the flag register.
- **SAHF**: Store (copy) AH register to low byte of flag register.
- **PUSHF**: copy flag register to TOS
- **POPF**: copy word at TOS to flag register.

2.Arithmetic Instructions

A. Addition Instructions

- **ADD**: Add specified byte to byte or specified word to word. ADD AX, BX
- **ADC**: add byte+ byte+ carry flag or word+ word+ carry flag .
- **INC**: increment specified byte or specified word by 1.INC AX
- **AAA**: ASCII Adjust after Addition
- **DAA**: Decimal (BCD) adjust after addition

B. Subtraction Instructions

- **SUB** :subtract byte from byte or word from word.
  SUB AX, BX
- **SBB**: Subtract byte and carry flag from byte or word and carry flag from word.
- **DEC**: Decrement the specified byte or specified word by 1.
- **NEG**: Negate-invert each bit of a specified byte or word and add 1(2's complement).
- **CMP**: compare 2 specified bytes or 2 specified words.
- **AAS**: ASCII Adjust after Subtraction.
- **DAS**: Decimal (BCD) adjust after subtraction

C. Multiplication instructions

- **MUL**: Multiply unsigned byte by byte or unsigned word by word.
- **IMUL**: Multiply signed…
- **AAM**: ASCII Adjust after multiplication

D. Division Instructions

- **DIV**: Divide unsigned word by byte or unsigned double word by word
- **IDIV**: Divide Signed…
- **AAD**: ASCII Adjust before division.
- **CBW**: Fill upper byte of word with copies of sign bit of lower byte.
- **CWD**: fill upper word of double word with sign bit of lower word.

3. Bit Manipulation Instructions

A. Logical Instructions

- **NOT**: Invert each bit of a byte or word
- **AND** :AND each bit in a byte or word with the corresponding bit in another byte or word.
- **OR**: OR…
- **XOR**: XOR…
- **TEST**: AND operands to update flags , but don’t change operands.

B. Shift Instructions

- **SHL/SAL**: Shift bits of word or byte left, put Zeros in LSB
- **SHR**: Shift bits of word or byte right, put 0s in MSBs.
- **SAR**: Shift bits of word or byte right , copy old MSB into new MSB.

C. Rotate Instructions

- **ROL**: Rotate bits of byte or word left, MSB to LSB and to CF.
- **ROR:** Rotate bits of byte or word right, LSB to MSB and CF.
- **RCL**: Rotate bits of byte or word left, MSB to CF and CF to LSB.
- **RCR**: Rotate bits of byte or word right, LSB to CF and CF to MSB.

4. String Instructions

- A String is a series of bytes or a series of words in sequential memory locations.
- A String often consists of ASCII character codes.
- B- used to indicate that a string of bytes is to b acted upon.
- W- used to indicate that a string of words is to b acted upon

- **REP**: An instruction prefix. Repeat following instruction until CX=0.
- **REPE/REPZ**: Repeat instruction until CX=0 or ZF<>1.
- **REPNE/REPNZ**: Repeat instruction until CX=0 or ZF=1.
- **MOVS/MOVSB/MOVSW**: move byte or word from 1string to another
- **COMPS/COMPSB/COMPSW**: compare 2 strings byte/word
- **INS/INSB/INSW**: input string byte or word from port.

- **OUTS/OUTSB/OUTSW**: output string byte/ word to port
- **SCAS/SCASB/SCASW**: scan a string , compare a string byte with a byte in AL or a string word with a word in AX.
- **LODS/LODSB/LODSW**: load string byte into AL or string word into AX.
- **STOS/STOSB/STOSW**: store byte from AL or word from AX into string

4. Program Execution transfer instructions

A. Unconditional transfer

- **CALL**: call a procedure (sub program), save return address on stack
- **RET**: return from procedure to calling program
- **JMP**: go to specified address to get next instruction

B. Conditional transfer

- **JA/JNBE**: jump on above or jump on not below or equal
- **JC**: jump on carry
- **JE/JZ**: jump on equal or zero.
- **JO**: jump on overflow…etc

C. Iteration control instructions

- **LOOP**: Loop through a series of instructions until CX=0.
- **LOOPE/LOOPZ** : Loop thru a seq. of instructions while ZF=1 and CX<>0.
- **LOOPNE/LOOPNZ** : Loop thru a seq. of instructions while ZF=0 and CX<>0.
- **JCXZ**: jump to specified adrs if CX=0

D. Interrupt instructions

- **INT**: Interrupt pgm execution, call service procedure.
- **INTO**: Interrupt pgm execution if OF=1.
- **IRET**: Return from interrupt service procedure to main program.

6.Processor Control Instructions

A. Flag set/clear Instructions

- **STC**: set CF to 1
- **CLC**: clear CF to 0.
- **CMC**: complement CF
- **STD**: set Direction Flag DF to 1(decrement string pointers)
- **CLD**: clear DF to 0.
- **STI**: set Interrupt enable flag to 1(enable INTR interrupt).
- **CLI**: clear interrupt enable flag to 0 (disbale INTR interrupt)

B. External Hardware synchronization instructions

- **HLT**: Halt (do nothing) until interrupt or reset.
- **WAIT**: wait (do nothing) until signal on the TEST pin is low.
- **ESC**: escape to external coprocessor such as 8087 or 8089
- **LOCK**: an instruction prefix. Prevents another processor from taking the bus while adjacent instruction executes.

C. No operation instructions

- **NOP**: no action except fetch and decode.

- https://www.slideshare.net/infinite2me/assembly-language-programmingunit-4?from_action=save