4 Assembly Language Programming (1)

> **Machine Language**: Binary, for CPU but not human beings.
>
> **Assembly Language**: Mnemonics（助记符） for machine code instructions.
>
> - *Low-level language*: deals with the internal structure of CPU;
> - Hard to program, poor portability but very efficient.
>
> **C, Python, ...**
>
> - *High-level language*: do not have to be concerned with the internal details of a CPU;
> - Easy to program, good portability but less efficient.
>
> **High-level language to Machine language**: Compiler.
>
> **Low-level language to Machine language**: Assembler.

Assembly Programming Language

- *Statements*:
    - Assembly language instructions: perform the real work of program (for *CPU*).
    - Directives (pseudo-instructions): Give instructions for the *assembler* program about how to translate the program into machine code.
- Consists of multiple segments;

Statement

$$[\text{label :}]\ \text{mnemonic}\ [\text{operands}][\text{; comment}]$$

- label is a reference to this statement: each label must be unique; letters, 0-9, (?), (.), (@), (_) and ($); first character cannot be a digit; less than 31 characters.
- (:) is needed if it is an instruction (not a pseudo one) otherwise omitted;
- (;) leads a comment, the assembler omits anything on this line following a semicolon.

MODEL Definition: Selects the size of the memory model. (*SMALL, MEDIDUM, LARGE, HUGE, TINY, COMPACT ...*)

Simplified Segment Definition

- Only three segments can be defined: `.CODE`, `.DATA`, `.STACK`;
- Automatically correspond to the CPU's `CS`, `DS`, `SS`.
- DOS determines the `CS` and `SS` segment registers automatically. `DS` (and `ES`) has to be manually specified.

Full Segment Definition

```
label SEGMENT
...
label ENDS
```

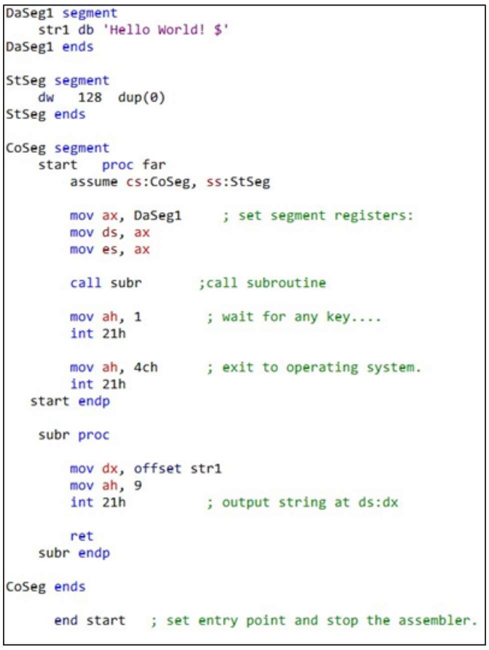
- You name those labels, and segments can be as many as needed.
- DOS assigns `CS` and `SS`; program assigns `DS` and `ES`.

Procedures Definition

```
label PROC [FAR|NEAR]
...
label ENDP
```

`NEAR` means the procedure is in the same code segment; `FAR` means the procedure is in another code segment. Entrance procedure should be `FAR`.

Program Execution

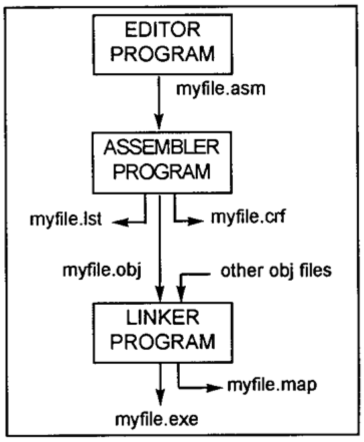


```
DaSeg1 segment
    str1 db 'Hello World! $'
DaSeg1 ends

StSeg segment
    dw   128  dup(0)
StSeg ends

CoSeg segment
    start   proc far
        assume cs:CoSeg, ss:StSeg

        mov ax, DaSeg1     ; set segment registers:
        mov ds, ax
        mov es, ax

        call subr       ;call subroutine

        mov ah, 1        ; wait for any key....
        int 21h

        mov ah, 4ch      ; exit to operating system.
        int 21h
   start endp

    subr proc

        mov dx, offset str1
        mov ah, 9
        int 21h          ; output string at ds:dx

        ret
    subr endp

CoSeg ends

      end start   ; set entry point and stop the assembler.
```

- Program starts from the entrance;
- Procedure caller（调用者） and callee（被调用）：

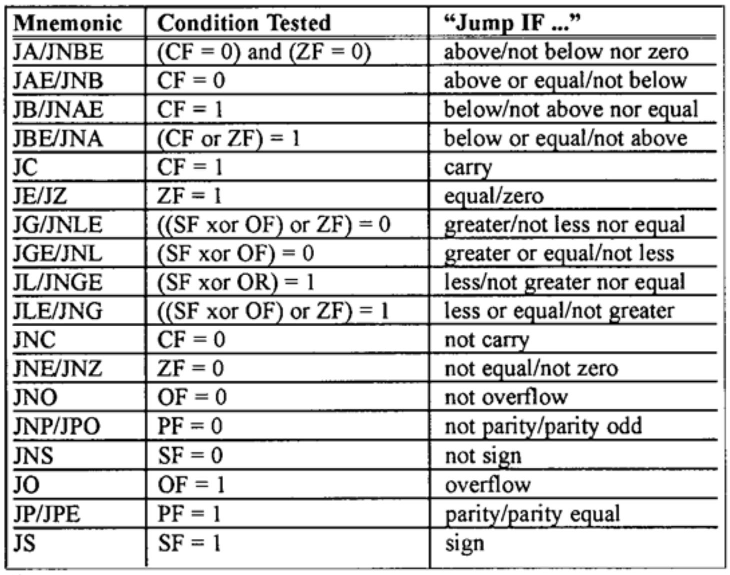
  ```
  CALL procedure
  ...
  RET
  ```

- Program ends whenever calls 21H interruption with $AH = 4CH$.

> **The Building Process of Assembly Language**
>
> 
>
>

Control Transfer Instructions

- `JUMP` instruction.
  - Conditional Jumps: Jumps according to the value of the flag register, usually SHORT jumps.



Mnemonic	Condition Tested	“Jump IF ...”
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

    - Unconditional Jumps: `JMP [SHORT|NEAR|FAR PTR] label`. (NEAR by default).
- `CALL` instruction.
    - Calling a NEAR procedure:
        - When we execute the `CALL` instruction, before jumping to the subroutine's instruction, the CPU will automatically put the address of the next instruction (`IP` value) in the stack.
        - After the subroutine ends, the `ret` instruction will pop the value from the stack and assign the value to the `IP` register, then CPU will execute the next instruction in the main process.
    - Calling a FAR procedure:
        - In the subroutine definition, we need to add a `far` in it.
        - In the `CALL` instruction, we need to add a `far` to distinguish it from a `NEAR` call. What's more, we need to
        - before jumping to the subroutine's instruction, the CPU will automatically put the address of the next instruction (both`CS` and `IP` value) in the stack.
        - After the subroutine ends, the `ret` instruction will pop the value from the stack and assign the values to the `IP` register and `CS` value, then CPU will execute the next instruction in the main process.
- Range:
    - `SHORT`: intra-segment, IP changed one-byte range (-128~127);
    - `NEAR`: intra-segment, IP changed two-bytes range (-32768~32767); Control is transferred within the same code segment.
    - `FAR`: inter-segment: CS and IP all changed; Control is transferred outside the current code segment.

Data Type & Definition

- CPU can process either 8-bit or 16-bit; if we want to handle 32-bit value, we need to separate it into different words.
- Directives
    - `ORG`: indicates the beginning of the offset address;

        > [*Example*] `ORG 10H`;

    - Define variables:
        - `DB`: allocate byte-size chunks;

> *[Example]*
>
> ```
> x DB 12
> y DB 23H, 48H
> Z DB 'Good Morning!'
> str DB "I'm good!"
> ```

Type	Explanation	Functionality
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

- `EQU`: define a constant

  > *[Example]* `NUM EQU 234`

- `DUP`: duplicate a given number of characters

  > *[Example]*
  >
  > ```
  > x DB 6 DUP(23H)
  > y DW 3 DUP(0FF10H)
  > ```
  >
  > Here an extra `0` in `0FF10H` indicates that `FF10H` is a number, not a variable.

Variables

- for variables, they may have names.
- Variable names have three attributes.
- Get the segment value of a variable: use `SEG` directive.
- Get the offset address of a variable: use `OFFSET` directive or `LEA` instruction.

  > *[Example]*
  >
  > ```
  > MOV AX, OFFSET time
  > LEA AX, time
  > ```

- Variable names have three attributes:
  - Segment value: logical address;
  - Offset address: logical address;
  - Type: how a variable can be accessed.

Labels

- Label Definition:
  - Implicitly: `AGAIN: ADD AX, 03423H`;
  - Use `LABEL` directive:

    > *[Example]*

> ```
> AGAIN LABEL FAR
> ADD AX, 03423H
> ```

- Labels have three attributes:
  - Segment value: logical address;
  - Offset address: logical address;
  - Type: range for jumps: NEAR, FAR

PTR Directive: temporarily change the type (range) attribute of a variable (label).

- To guarantee that both operands in an instruction match;

  > [*Example*]
  >
  > ```
  > DATA1 DB 10H,20H,30H
  > DATA2 DW 4023H, 0A845H
  > MOV BX, WORD PTR DATA 1 ; 2010H -> BX
  > MOV AL, BYTE PTR DATA 2 ; 23H -> AL
  > MOV WORD PTR [BX], 10H; [BX], [BX+1] <- 0010H
  > ```

- To guarantee that the jump can reach a label.

  > [*Example*]
  >
  > ```
  > JMP FAR PTR aLabel
  > ```

.COM Executable: one segment in total, put data and code all together, less than 64KB. Use `JUMP` to skip the data lines.