**2.Main Assembler Pseudo-Instructions**

Commands(orders, instructions) for assembler, necessary for the proper translation of the program and for facilitating the programmer’s job.

Do not generate machine code, they only instruct the assembler how to deal with things

-declararea unui segment, unei procedure si a datelor

Assume

1. **Pseudo-instructions to define segments**

Segment= identified with a name and a class

= receives a series of attributes -> specify the relation between segments

DEF : **segment\_name** SEGMENT [**align\_type**][**combine\_type**][‘**class**’]

;……..

Segment\_name ENDS;

**Segment’s nam**e is associated with a value corresponding to the segment position in memory

**Align\_type** =segment’s alignment types

1. Para – paragraph alignment (16 bytes multiple)
2. Byte- byte alignment
3. Word-word alignment
4. Page – page alignment( 256 byte multiple)

**combine\_type** controls the order that segments with the same name are written out to the object code file produced by the assembler. To specify the combine type you use one of the keywords **public, stack, common, memory**, or at. Memory is a synonym for public provided for compatibility reasons; you should always use public rather than memory. Common and at are advanced combine types that won't be considered in this text. The stack combine type should be used with your stack segments. The public combine type should be used with most everything else.

1. **Public-**concatenation
2. **Common**-advanced type
3. **At** – segment’s load having the address expr \* 16
4. **Stack** – curr segment= stack segment
5. **Memory** – segment’s location = the last segment from the program( highest in physical memory)

**Class = segment’s class**

* The link-editor continually arranges the segments having the same in order of their appeareance
* Types of classes

1. Code
2. Data
3. Constant
4. Memory
5. Stack
6. **The designation of the active segment**

* In a program more segment can be defined
* The assembler verifies whether the data or addresses may be reached with instructions

The segment register having a certain content

for a proper workflow the address of the active segments must be communicated to the assembler ( till segment register contains the address of each segment)

ASSUME <**seg\_reg**>: <**seg\_name**>,<**seg\_reg**>: <**seg\_name**>……

**Seg\_reg** = the segment register

**Seg\_name =**  the segment which will be active( nothing ->don’t associate a segment to the seg register)

Example : assume CS:prg, DS:date, ES :date2

**REMARKS**

**1.**The pseudo instruction **does not load** the segment register, it communicates the assembler when the symbols are defined

2. DS- loaded

ASSUME DS: data\_seg\_name

MOV AX, data\_seg\_name

MOV DS,AX

1. CS – not initialized, but activated (ASSUME) before the 1st label

**3.Memory location reservation**

-data – defined in a data segment

Def: & allocate space in memory

<name> <type> [expression list][<factor> DUP (<expression list>)]

**Name = identifier**

**Type =**  symbol’s type

1. **Db – 1 byte**
2. **Dw 1 word space (2 bytes)**
3. **DD double word space ( 4 bytes)**
4. **DQ quadruple word space ( 8 bytes)**
5. **DT – 10 bytes space**

**Expressions list –** the memory location is initialized with the result of this expression

**-“?” = allocate with no initialization**

**Factor =**  a constant -> how many times the list after DUP is repeated.

Examples :

DAT DB 45

Date DB ‘abcdefghi’

Bb db 20 dup (?)

1. **Other possibilities for defining symbols**
2. Def of constants

<**name** > EQU <expression>

<name> = <expression>

* The name symbol will be replaced with the value’s expression
* ‘=’ allows redefinition if constant. While EQU does not.

1. Declaration of labels

<name> LABEL <type>

Name- value= segment where it is defined

Offset= offset of the 1st init for data assignment or other instr which flow

Type – Byte, word , dword , qword, tbyte, name of a structure,near,far

-: after the name => near

1. **Current location counter( CLC ) modification**

ORG <expression>

ORG 100h

Org $=2 ; $= CLC’s current value

The CLC will get the expression’s value

1. **Procedure definition= a sequence of instructions which ends with RET and is called with CALL**

Def: < procedure\_name> proc <[NEAR],FAR>

; instructions

<procedure\_name> ENDP

Remarks:

1. The declaration of the procedure, nor the ENDP, do not generate any machine
2. The user must assure the returning with RET
3. The same proc may not be called as FAR and as NEAR
4. Nested proc are allowed