## Assembly Language Syntax and Program Structure

Introduction

A processor can directly execute a machine language program. Though it is possible to program directly in machine language, assembly language uses mnemonics to make programming easier.

An assembly language program uses mnemonics to represent:

- symbolic instructions and
- the raw data that represent variables and constants.

A machine language program consists of:

- a list of numbers representing the bytes of machine instructions to be executed and
- data constants to be used by the program.

## Assembly Language Syntax

An assembly language program consists of statements. The syntax of an assembly language program statement obeys the following rules:

- Only one statement is written per line
- Each statement is either an instruction or an assembler directive
- Each instruction has an opcode and possibly one or more operands
- An opcode is known as a mnemonic
- Each mnemonic represents a single machine instruction
- Operands provide the data to work with.

**Program Statement:**

The general format for an assembly language program statement is as follows:

**Name mnemonic operand(destination), operand(source) ; comment**

**Name Field:**

This field is used for:

- instruction label: if present, a label must be followed by a colon (**:**)
- procedure names
- variable names.

```
Examples of Name Fields
here:   MOV AX,0867H    ; Statement line with a label field
        JNC here        ; Statement with opcode and one operand
        INT 03H
SUM PROC NEAR           ; Procedure definition
```

**Operation Field:**

- This field consists of a symbolic operation code, known as opcode
- The opcode describes the operationâ€™s function
- Symbolic opcodes are translated into machine language opcode.

**Operand Field:**

This field specifies data to be acted on. It may have one, two or no operands at all.

Examples of instructions with different operand fields

```
NOP       ; Instruction with no operand field
INC AX    ; Instruction with one operand field
ADD AX, 2 ; Instruction with two operand field
```

**Comment Field:**

- A semicolon marks the beginning of a comment
- A semicolon in the beginning of a line makes it all a comment line
- Good programming practice dictates the use of a comment on almost every line.

**Key rules for the use of comments:**

- Do not say something that is obvious
- Put instruction in context of program

Examples of good and bad Comments

```
MOV CX, 0  ; Move 0 to CX. (This is not a good comment.)
MOV CX, 0  ; CX counts terms, initially set to 0. (This is a good comment.)
```

**Assembler Directives:**

Pseudoinstructions or assembler directives are instructions that are directed to the assembler. Assembler directives affect the generated machine code, but are not translated directly into machine code. Directives can be used to declare variables, constants, segments, macros, and procedures as well as supporting conditional assembly.

In general, a directive:

- contains pseudo-operation code,
- tells the assembler to do a specific thing, and
- is not translated into machine code.

## Naming Conventions

Assembly language imposes some rules on how names are assigned to labels, variables, procedures and macros. It is the assembler's function to translate those names into memory addresses. Note that naming conventions used in this course are related to the MASM 6.11 assembler used in the lab.

**Names:**

A name is used to identify a label, a variable, a procedure or a macro. Here are the general rules on the use of names:

- A name is between 1 and 31 characters in length
- A name may include letters, numbers and special characters, such as **? . @ _ $ %**
- A name should not begin with a digit
- A name may begin with a letter or a special character
- If a period (**.**) is used, it must be the first character
- Names are not case sensitive.

Below are examples of legal and illegal names.

**Examples of legal names**

- COUNTER1
- @character
- SUM_TOTAL
- $1000
- Done?
- .TEST

**Examples of illegal names**

- TWO WORDS
- 2abc
- A45.28
- You&Me
- A+B

## Structure of an Assembly Language Program

**Program Structure:**

A program has always the following general structure:

**Structure of an Assembly Language Program**

```
.model small                ;Select a memory model

.stack stack_size     ;Define the stack size

.data

                            ; Variable and array declarations
                            ; Declare variables at this level
.code
main proc

                            ; Write the program main code at this level

main endp


;Other Procedures

                            ; Always organize your program
                            ; into procedures


end main                    ; To mark the end of the source file
```

**Title directive:**

The title directive is optional and specifies the title of the program. Like a comment, it has no effect on the program. It is just used to make the program easier to understant.

**The Model directive:**

The model directive specifies the total amount of memory the program would take. In other words, it gives information on how much memory the assembler would allocate for the program. This depends on the size of the data and the size of the program or code.

**Segment directives:**

Segments are declared using directives. The following directives are used to specify the following segments:

- stack
- data
- code

**Stack Segment:**

- Used to set aside storage for the stack
- Stack addresses are computed as offsets into this segment
- Use: .stack followed by a value that inicates the size of the stack

**Data Segments:**

- Used to set aside storage for variables.
- Constants are defined within this segment in the program source.
- Variable addresses are computed as offsets from the start of this segment
- Use: .data followed by declarations of variables or defintions of constants.

**Code Segment:**

- The code segment contains executable instructions macros and calls to procedures.
- Use: .code followed by a sequence of program statements.

**Example:**

Here is how the "hello world" program would look like in assembly:

**The hello world program in Assembly**

```
.model small
.stack 200

.data
      greeting db 'hello world !',13,10,'$'

.code
mov ax,@data
mov ds,ax
mov ah,9
mov dx,offset greeting
int 21h
mov ah,4ch
int 21h
end
```

## Memory Models

The memory model specifies the memory size assigned to each of the different parts or segments of a program. There exist different memory models for the 8086 proceessor

**The .MODEL Directive**

The memory model directive specifies the size of the memory the program needs. Based on this directive, the assembler assigns the required amount of memory to data and code.

Each one of the segments (stack, data and code), in a program, is called a *logical segment* . Depending on the model used, segments may be in one or in different physical segments.

In MASM 6.X, segments are declared using the .MODEL directive. This directive is placed at the very beginning of the program, or after the optional title directive.

| **MODEL directive** |
| --- |
| .MODEL memory_model<br>Where memory_model can be: |

- TINY
- SMALL
- COMPACT
- MEDIUM
- LARGE or
- HUGE.

**TINY Model:**

In the TINY model both code and data occupy one physical segment. Therefore, all procedures and variables are by default addressed as NEAR, by pointing at their offsets in the segment. On assembling and linking a source file, the tiny model automatically generates a com file, which is smaller in size than an exe file.

**SMALL Model:**

In the SMALL model all code is placed in one physical segment and all data in another physical segment.
In this model, all procedures and variables are addressed as NEAR by pointing to their offsets only.

**COMPACT Model:**

In the COMPACT model, all elements of code (e.g. procedures) are placed into one physical segment. However, each element of data can be placed by default into its own physical segment. Consequently, data elements are addressed by pointing both at the segment and offset addresses. In this model, all code elements (procedures) are addressed as NEAR and data elements (variables) are addressed as FAR.

**MEDIUM Model:**

The MEDIUM model is the opposite of the compact model. In this model data elements are treated as NEAR and code elements are addressed as FAR.

**LARGE Model:**

In the LARGE model both code elements (procedures) and data elements (variables) are put in different physical segments. Procedures and variables are addressed as FAR by pointing at both the segment and offset addresses that contain those elements. However, no data array can have a size that exceeds one physical segment (i.e. 64 KB).

**HUGE Model:**

The HUGE memory is similar to the LARGE model with the exception that a data array may have a size that exceeds one physical segment (i.e. 64 KB).

The following table summarizes the use of models and the number and sizes of physical segments that are used with each of the models.

| Memory Model | Size of Code | Size of Data |
|---|---|---|
| TINY | Code + Data < 64KB | Code + data < 64KB |
| SMALL | Less than 64KB | Less than 64KB |
| MEDIUM | Can be more than 64KB | Less than 64 KB |
| COMPACT | Less than 64KB | Can be more than 64KB |
| LARGE* | Can be more than 64K | Can be more than 64KB |
| HUGE** | Can be more than 64K | Can be more than 64KB |

Table 1: Memory Models

(*) For the LARGE model, the largest arrays size can not exceed 64 KB.(br) (**)For the HUGE model, an array may have a size greater than 64 KB and hence can span more than one physical segment.

**Use of memory models:**

The amount of data that has to be manipulated and code that needs to be written are the mojor factors in determining the choice of an appropriate model. The following are guidelines to help chose the right model for a program.

For a small fast program that operates on small quantities of data, the SMALL or TINY models are the most suitable ones. These models allow up to 64K of memory (i.e. one single physical segment), but the executable code is fast since only near references are used in the calculation of addresses. The only difference between these two models is that the TINY model generates a .COM module in which far references cannot be used, whereas the SMALL model generates a .exe module.

For very long programs that require more than one code segment and operate on large amounts of data which would require more than one data segment, the LARGE and HUGE models are most appropriate.

## Instructions

**Definition:**

An instruction in assembly language is a symbolic representation of a single machine instruction. In its simplest form, an instruction consists of a mnemonic and a list of operands.

- A mnemonic is a short alphabetic code that assists the CPU in remembering an instruction. This mnemonic can be followed by a list of operands
- Each instruction in assembly language is coded into one or more bytes
- The first byte is generally an OpCode, i.e. a numeric code representing a particular instruction
- Additional bytes may affect the action of the instruction or provide information about the data needed by the instruction.

**Instruction Types:**

There exist around 150 instructions for the 8086 processor. These instructions may be classified into different classes. The following table summarizes the different classes of instructions, together woth examples of each class.

| Instruction Type | Definiton | Examples |
|---|---|---|
| Data transfer instructions | Transfer information between registers and memory locations or I/O ports | MOV, XCHG, LEA,PUSH,POP |
| Arithmetic instructions | Perform arithmetic operations on binary or binary-coded-decimal (BCD) numbers | ADD, SUB, INC, DEC |
| Bit manipulation instructions | Perform shift, rotate, and logical operations on memory locations and registers | SHL, SHR, SAR, ROL |
| Control transfer instructions | Control sequence of program execution. Include jumps and procedure transfers | JL, JE, JNE, JGE |
| String handling instructions | Move, compare, and scan strings of data | OVSB, MOVSW, CMPS, CMPSB |
| Processor control instructions | Set and clear status flags, and change the processor execution state | STC, STD, STI, CLC |
| Interrupt instructions | Interrupt processor to service a specific condition | INT, INTO, IRET |
| Miscellaneous instructions | | NOP, WAIT |

Table 2: 8086 Instructions

**Instruction Semantics:**

The following rules have to be strictly followed in order to write correct code.

1 - Both operands have to be of the same size:

| Instruction | Correct | Reason |
|---|---|---|
| MOV AX, BL | No | Operands of different sizes |
| MOV AL, BL | Yes | Operands of same sizes |
| MOV AH, BL | Yes | Operands of same sizes |
| MOV BL, CX | No | Operands of different sizes |

Table 3: Register-Register Transfer Instructions

2 - Both operands cannot be memory operands simultaneously:

| Instruction | Correct | Reason |
|---|---|---|
| MOV i , j | No | Both operands are memory variables |
| MOV AL, i | Yes | Move memory variable to register |
| MOV j, CL | Yes | Move register to memory variable |

Table 4: Memory-Register Transfer Instructions

3 - First operand, or destination, cannot be an immediate value:

| Instruction | Correct | Reason |
|---|---|---|
| ADD 2, AX | No | Move register to constant |
| ADD AX, 2 | yes | Move constant to register |

Table 5: Constant to Register Transfer Instructions

## Directives

Assembler directives are special instructions that provide information to the assembler. Assembler directives do not generate code.
Segment directives, equ, assume, and end mnemonics are all directives. These are not valid 80x86 instructions.

Directives may be divided into the following classes:

- Pseudo-Opcode directive or Processor Code Generation Directives
- Memory Model Directives
- Segment Definition Directives
- Segment Ordering Directives
- Linkage Directives
- Data Allocation Directives
- Logical and Bit Oriented Directives
- Macro Definition Directives
- Program Listing and Documentation Directives
- Conditional Assembly Directives
- User Message Directives
- Predefined Equates
- Radix Specifiers
- Other Operators and Directives

**Pseudo-Opcode Directives:**

A pseudo-opcode is a message to the assembler just like an assembler directive. However, a pseudo-opcode will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and tbyte. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

At this level only the most commonly used directives are briefly presented.

**Memory Model Directives:**

These are already seen in the section related to models. The following table summarizes the main points about segment directives.

| Directive | Effect |
|---|---|
| .MODEL model | defines memory model to be one of the following: SMALL, COMPACT, MEDIUM, LARGE or HUGE; must be used prior to any other segment directive |
| .CODE [name] | starts code segment; must follow .MODEL directive |
| .DATA | starts a near data segment for initialized data must follow .MODEL directive; |
| .STACK [size] | indicates start of stack segment named 'STACK' with size indicating number of bytes to reserve. |

Table 6: Model and Segment directives

**Segment Definition Directives:**

| Directive | Effect |
|---|---|
| name PROC [NEAR \| FAR] | defines procedure; NEAR/FAR has .MODEL default |
| name ENDP | ends procedure 'name' |
| PUBLIC name[,name...] | makes symbol 'name', which could be a variable or |
| END [name] | marks end of source module and sets program start address (CS:IP) if 'name' is present a procedure available to other modules. |

Table 7: Segment Definition directives

**Data Allocation Directives:**

| Directive | Effect |
|---|---|
| [name] DB init[,init...] | define byte |
| [name] DW init[,init...] | define word (WORD, 2 bytes) |
| [name] DD init[,init...] | define double word (DWORD, 4 bytes) |
| [name] DF init[,init...] | define far word (FWORD, 386, 6 bytes) |
| [name] DQ init[,init...] | define quad word (QWORD, 8 bytes) |
| [name] DT init[,init...] | define temp word (TBYTE, 10 bytes) |
| count DUP(init[,init...]) | duplicate 'init' 'count' times; DUP can be |

Table 8: Data Allocation directives

**Predefined Equates:**

These are available only if simplified segments are used.

| Directive | Effect |
|---|---|
| @code | contains the current code segment |
| @codesize | 0 for small and compact; 1 for large medium and huge |
| @datasize | 0 for small and medium; 1 for compact ;2 for large |
| @data | contains segment of define by .DATA |
| @stack | contains segment of define by .STACK |

Table 9: Equates directives

**Radix Specifiers:**

| Directive | Effect |
|---|---|
| .RADIX expr | sets radix [2..16] for numbers (dec. default) |
| .RADIX B | binary data specifier |
| .RADIX Q | octal data specifier |
| .RADIX O | octal data specifier |
| .RADIX D | decimal data specifier (default) |
| .RADIX H | hex |

Table 10: Radix Specifiers directives

**Macro Definition Directives:**

| Directive | Effect |
|---|---|
| name MACRO [parm[,parm...]] | defines a macro and it's parameters |
| ENDM | terminates a macro block |
| LOCAL name[,name...] | defines scope symbol as local to a macro |
| EXITM | exit macro expansion immediately |
| REPT expr | repeats all statements through ENDM statement for 'expr' times |

Table 11: Macro Definition directives

## Writing a Program

**How to write an assembly language program:**

These are the steps that should be followed for writing an assembly language program:

- First, define the problem
- Write the algorithm
- Translate into assembly mnemonics
- Test and Debug the program in case of errors

The translation phase consists of the following steps:
- Define type of data the program will deal with
- Write appropriate instructions to implement the algorithm

**Problem:**

In this example you are asked to write a program that calculates the average of two numbers.

**Write the algorithm:**

- add maximum temp and minimum temp to get the sum
- divide sum by 2 to get average

**Translate into Assembly:**

- what is our data?
- what are our initialization steps?

Here is the listing of the program as it should be given to the computer. It should of course first be assembled and linked.

```
Program that calculates the average of 2 numbers
.model small
.data
        max_temp DB 92h
        min_temp DB 52h
        avg_temp DB ?
.code
```

```
.startup

        mov ax,@data
        mov ds,ax
        mov al, max_temp
        add al, min_temp
        mov ah, 00h
        adc ah, 00h
        mov bl, 02h
        div bl
        mov avg_temp, al
.exit
end
```

**Debugging Tips:**

As a general rule, no program is supposed to work as one would expect when it is run for the first time. Therefore, a debugging phase, in which you have to find and correct errors is necessary. Here are some tips to help you go through the debugging phase.

- Be sure to work out an algorithm firstâ€¦ not as you go along
- write and test sections of a larger program, rather than waiting until you think youâ€™re done to do any testing
- make sure you coded according to a correct algorithm
- add a set of eyes or watches
- use a debugger such as the codeview that you use in the lab.
- single step through code
- strategically place breakpoints