**Assemblers:**

• Programmers found it difficult to write or red programs in machine language. In a quest for a convenient language, they began to use a mnemonic (symbol) for each machine instructions which would subsequently be translated into machine language.

• Such a mnemonic language is called Assembly language.

• Programs known as Assemblers are written to automate the translation of assembly language into machine language. Assembly language program Machine language program

• **Fundamental functions:**

1. Translating mnemonic operation codes to their machine language equivalents.

2. Assigning machine addresses to symbolic tables used by the program.

**Elements of Assembly Language Programming**

-Integer constants

-Integer expressions

-Character and string constants

-Reserved words and identifiers

-Directives and instructions

-Labels

-Mnemonics and Operands

-Comments

**Integer Constants & Operators**

• Optional leading + or – sign

• binary, decimal, hexadecimal, or octal digits

• Common radix characters:

• h – hexadecimal

• d – decimal

• b – binary

• r – encoded real

 Examples: 30d, 6Ah, 42, 1101b

• Hexadecimal beginning with letter: 0A5h

**Character and String Constants**

1."Enclose character in single or double quotes

     'A', "x"

    ASCII character = 1 byte

2. Enclose strings in single or double quotes

   "ABC“ 'xyz'

3. Each character occupies a single byte

4. Embedded quotes:

   'Say "Goodnight," Gracie'

**Reserved Words and Identifiers**

•Reserved words (Appendix D) cannot be used as identifiers

•Instruction mnemonics, directives, type attributes, operators, predefined symbols

•Identifiers

•1-247 characters, including digits

•case insensitive (by default)

•first character must be a letter, \_, @, or $

•Directives

•command understood by the assembler

•not part of Intel instruction set

•case insensitive

**Directives**

•Commands that are recognized and acted upon by the assembler

•Not part of the Intel instruction set

•Used to declare code, data areas, select memory model, declare procedures, etc.

•Different assemblers have different directives

•NASM != MASM, for example

**Instructions**

•Assembled into machine code by assembler

•Executed at runtime by the CPU

•Member of the Intel IA-32 instruction set

•Parts

•Label

•Mnemonic

•Operand

•Comment

**Labels**

•Act as place markers

•marks the address (offset) of code and data

•Follow identifer rules

•Data label

•must be unique

•example: **myArray** (not followed by colon)

•Code label

•target of jump and loop instructions

**Mnemonics and Operands**

•Instruction Mnemonics

•"reminder"

•examples: MOV, ADD, SUB, MUL, INC, DEC

•Operands

•constant (immediate value)

•constant expression

•register

•memory (data label)

**Comments**

•Comments are good!

•explain the program's purpose

•when it was written, and by whom

•revision information

•tricky coding techniques

•application-specific explanations

•Single-line comments

•begin with semicolon (;)

•Multi-line comments

•begin with COMMENT directive and a programmer-chosen character

•end with the same programmer-chosen character

**Design of assembler**

specify the problem

1.specify the data structure

2.define format of data structure

3.specify algorithm

4.look for modularity

5. repeat 1 ~ 5 on modules

The design of assembler can beto perform the following:

Scanning (tokenizing)

Parsing (validating the instructions)

Creating the symbol table

Resolving the forward references

Converting into the machine language

**The design of assembler in other words:**

-Convert mnemonic operation codes to their machine language equivalents

-Convert symbolic operands to their equivalent machine addresses

-Decide the proper instruction format Convert the data constants to internal machine

representations

-Write the object program and the assembly listing

We need to identify the algorithms and the various data structures to be used. According to the above required steps for assembling the assembler also hasto handle

*assembler directives* these do not generate the object code but directs the assembler to perform certain operation.

These directives are:

•SIC Assembler Directive:

START:Specify name & starting address.

END:End of the program, specify the first execution instruction.

In this section, two alternatives to the standard two-pass assembler logic is discussed.

They are:

Single Pass Assembler

Multipass Assemble

3.2.1 SINGLE PASS ASSEMBLER

These assemblers are used when it is necessary or desirable to avoid a second pass over the source program. The main problem in designing the assembler using single pass was to resolve forward references. One-pass assemblers could produce object codes either in memory or to external storage. One-pass assemblers usually need to modify object code already generated, so whether object code is stored in memory or external storage imposes different considerations on assembler design.

Based on this one-pass assemblers can be classified into two types:

1. One that produces object code directly in memory for immediate execution (Loadand-go assemblers).

2. One pass assembler generating object code for later execution.

1. **Load-and-Go Assembler Load-and-go assembler** generates their object code in memory for immediate execution. Since no object program is written out, no loader is needed. It is useful in a system with frequent program development and testing. Since the object program is produced in memory, the handling of forward references becomes less difficult.

**Working of One pass assembler (Load and Go Assembler)** In load-and-Go assemblers when a forward reference is encountered :

-Omits the operand address if the symbol has not yet been defined(placess 000 at the operand addresses position)

- Enters this undefined symbol into SYMTAB and indicates that it is undefined

- Adds the location at which the operand is referenced to a list of forward references associated with the SYMTAB entry

At the end of the program, repo•- When the definition for the symbol is encountered, scans the reference list and inserts the address. rts the error if there are still SYMTAB entries indicated undefined symbols(\* indicates undefined).

- When the END statement is encountered, search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error. In short, whenever any undefined symbol is encountered it will insert into SYMTAB as a new entry and indicate that it is undefined and also adds the location at which the operand is referenced as a linked list associated with that SYMTAB entry. When the definition for the symbol is encountered, scans the reference list and inserts the address in proper location.

**2. One pass assembler generating object code for later execution.**

In this type of one pass assembler, the generated object program is stored in external storage (e.g.,files on disks). So random updates to operands target addresses(as in load-andgo load-and- assemblers do) are not permitted. For any symbol involved in forward references, once the target address of the symbol is identified, additional text records must be generated to overwrite those previously omitted target addresses. Records must be loaded in the same order as they appear in the object program. Actually, the handling of forward references are jointly done by the assembler and the linking loader.

One pass assembler which generates object code unlike load and go assembler operates in the following fashion: -- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.

- Forward references are entered into lists as in the load-and-go assembler

-When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.

- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition

**MULTI PASS ASSEMBLER**

- For a two pass assembler, forward references in symbol definition are not allowed:

ALPHA EQU BETA

BETA EQU DELTA

DELTA RESW 1

- Here the problem is, the symbol BETA cannot be assigned a value when it is encountered during Pass 1 because DELTA has not yet been defined. Hence ALPHA cannot be evaluated during Pass 2. So that the symbol definition must be completed in pass 1.

- The general solution for this type of forward references is to use a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.

- It is not necessary for such an assembler to make more than 2 passes over the entire program.

- The portions of the program that involve forward references in symbol definition are saved during Pass 1.Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2. Implementation of Multipass Assembler

- For a forward reference in symbol definition,

we store in the SYMTAB:

o The symbol name

o The defining expression

o The number of undefined symbols in the defining expression

- The undefined symbol (marked as \*) associated with a list of symbols depend on this undefined symbol.

- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

- The portions of the program that involve forward references in symbol definition are saved during Pass 1.Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

**Assembler Design Criteria**

• Design Specification of an assembler

 – Four step approach to develop a design specification

1) Identify the information necessary to perform a task

2) Design a suitable data structure to record the information

3) Determine the processing necessary to obtain and maintain

   the information.

4) Determine the processing necessary to perform the task

References

•<https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/assembler/idalangref_firstpass.html>

•<https://www.geeksforgeeks.org/introduction-of-assembler/>

•<http://www.csie.ntnu.edu.tw/~ghhwang/course_slices/system_software/Chapter2.pdf>