

System Programming

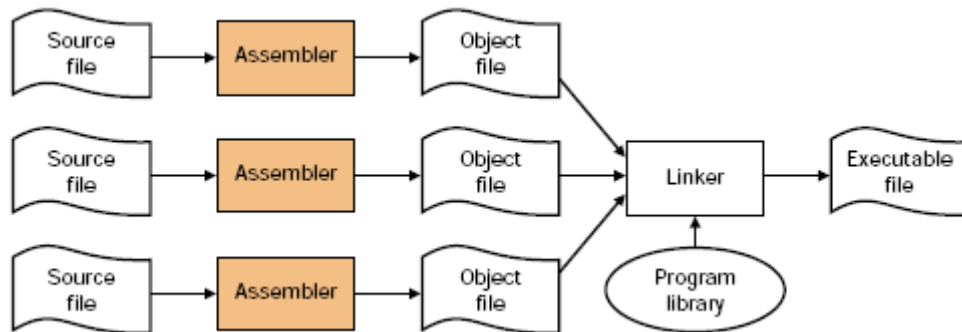
Second Class

مدرس مساعد: منال مطلوب

CHAPTER THREE (Assemblers)

Basic Assembler functions:

The assembler is responsible for translating the assembly language program into machine code. When the source language is essentially a symbolic representation for a numerical machine language, the translator is called an **assembler** and the source language is called an assembly language. A pure assembly language is a language in which each statement produces exactly one machine instruction figure below show the assembler functions.



1. Translate mnemonic opcodes to machine language.
2. Convert symbolic operands to their machine addresses.
3. Build machine instructions in the proper format.
4. Convert data constants into machine representation.
5. Error checking is provided.
6. Changes can be quickly and easily incorporated with a reassembly.
7. Variables are represented by symbolic names, not as memory locations.

Assembly language statements are written one per line. A machine code program thus consists of sequence of assembly language statements, where each statement contains a mnemonic. Each line of an assembly language program is split into four fields, as show below:

LABEL OPCODE OPERAND COMMENTS

Label: it is an identifier and optional field. Labels are used extensively in programs to reduce reliance upon programmers remembering where data or code is located. The maximum length of label differs between assemblers. Some accept upto 32 characters long, other only four characters. A label when declared is suffixed by a colon and begins with a valid character (A..Z)

For example

START: LDAA #24 H

Here, the label START is equal to the address of instruction
LDAA #24 H

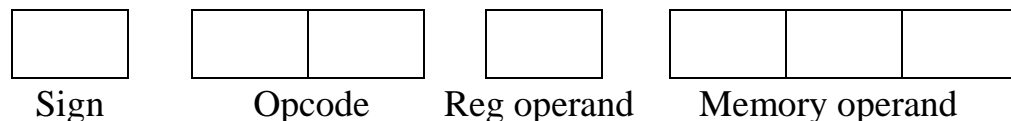
An advantage of using labels is that inserting or rearranging code statements do not necessitate reworking actual machine instructions.

Opcode: this field contains a mnemonic. Opcode stands for operation code i.e. a machine code instruction. The opcode may also require additional information i.e. operands.

Operand: this field consists of additional information or data that the opcode requires. In certain types of addressing modes, the operand is used to specify

- Constants or labels
- Immediate data
- Data contained in another accumulator or register.
- An address

The figure below shows the machine instruction format.



Assembly Language Statements

It consists of three types of statements:

- 1- Imperative
- 2- Declaration
- 3- Assembler

Imperative statement: it indicates an action to be performed during the execution of the assembled program. Imperative programs focus on how to solve a problem popular imperative languages such as C and Pascal are based on side effects on memory. Each imperative statement translates into one machine instruction.

Declaration statement: it focus on what the problem is and leave solution mechanism up to the language implementation. Within the declarative paradigm, most notable are the function languages such as lisp. Declarative languages are typically quite abstract and hence can be harder to implement efficiently.

Assembler directive: assembler directives instruct the assembler to perform certain actions during the assembly of a program. They can be used to declare variables, create storage space for results, to declare constants. The following assembler directives are used in the program:

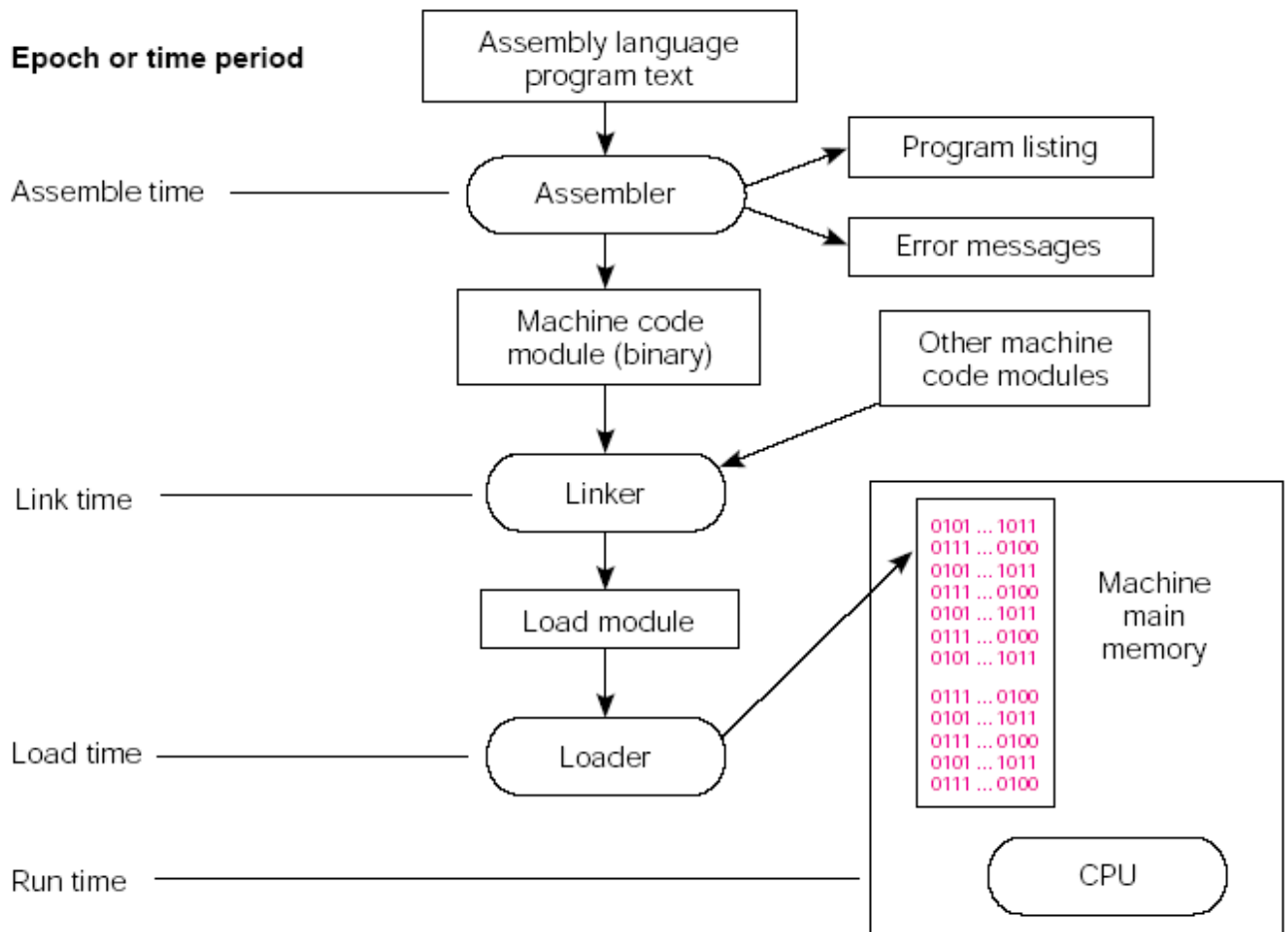
- 1- START-specify name and starting address for the program. START <constant>
- 2- END-indicate the end of the source program and specify the first executable instruction in the program. END [<operand spec>]
- 3- BYTE-generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
- 4- WORD-generate one word integer constant.

Advantages of assembly language

1. reduced errors.
2. Faster translation times.
3. Changes could be made easier and faster.

Disadvantages

1. Many instructions are required to achieve small tasks.
2. Source programs tend to be large and difficult to follow.
3. Programmer requires knowledge of the processor architecture and instruction set.
4. Programs are machine dependent, requires complete rewrites if the hardware is changed.



Directives Assembler

1. Directives are commands to the Assembler
2. They tell the assembler what you want it to do, e.g.
 - a. Where in memory to store the code
 - b. Where in memory to store data
 - c. Where to store a constant and what its value is
 - d. The values of user-defined symbols

Object File Format

Assemblers produce object files. An object file on Unix contains six distinct sections (see Figure 3):

- The *object file header* describes the size and position of the other pieces of the file.
- The *text segment* contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.
- The *data segment* contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The *relocation information* identifies instructions and data words that depend on absolute addresses. These references must change if portions of the program are moved in memory.
- The *symbol table* associates addresses with external labels in the source file and lists unresolved references.
- The *debugging information* contains a concise description of the way in which the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

The assembler produces an object file that contains a binary representation of the program and data and additional information to help link pieces of a program. This relocation information is necessary because the assembler does not know which memory locations a procedure or piece of data will occupy after it is linked with the rest of the program. Procedures and data

from a file are stored in a contiguous piece of memory, but the assembler does not know where this memory will be located. The assembler also passes some symbol table entries to the linker. In particular, the assembler must record which external symbols are defined in a file and what unresolved references occur in a file.

Macros

Macros are a pattern-matching and replacement facility that provide a simple mechanism to name a frequently used sequence of instructions. Instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

The 2-Pass Assembly Process

- **Pass 1:**
 1. Initialize location counter (assemble-time "PC") to 0
 2. Pass over program text: enter all symbols into symbol table
 - a. May not be able to map all symbols on first pass
 - b. Definition before use is usually allowed
 3. Determine size of each instruction, map to a location
 - a. Uses pattern matching to relate opcode to pattern
 - b. Increment location counter by size
 - c. Change location counter in response to ORG pseudos
- **Pass 2:**
 1. Insert binary code for each opcode and value
 2. "Fix up" forward references and variable-sizes instructions
 - Examples include variable-sized branch offsets and constant fields