ASSEMBLER - LINKER - LOADER

Submitted by:

11010106- Arihant Sethia

11010125- Yoshitha Kuntumalla

11010157- Rakshita Jain

Under the guidance of:

Prof. Santosh Biswas

INTRODUCTION:

An assembly language is a machine dependent, low level programming language which is specific to a certain computer system. It has three basic features – Mnemonic operation codes, Symbolic operands, and Data declarations. Our software aims at converting simple assembly language code defined on an instruction set into 8085 assembly code, linking different files and their variables and loading it in appropriate location in the memory as defined by the user. It involves three basic functions- Assembling, Linking and Loading.

An Assembler converts an assembly language code to an object file or machine language format. An assembler creates object code by translating assembly instruction mnemonics into op-codes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Along with this it also includes features of converting macros (a unit of specification for program generation through expansion).

A linker takes one or more object files generated by an assembler and combine them into a single executable program. It allows object files to use symbols defined in other modules or other files. The linker processes a set of object modules to produce a ready-to-execute program form, which we will call a binary program.

A Loader is responsible for loading programs for the purpose of execution. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable files, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running.

MODULES:

INPUT: The software has an instruction set of its own defined in 8085 assembly instruction set as the base. The user can write his program using the 8085 assembly instruction set as well as the pre-defined instruction set of this software.

PASS1 (ASSEMBLER): If any Macros are present in the code, they get replaced and also the op-codes used from the pre-defined instruction set with the 8085 assembly code it is written in. At the end of the pass1 a macro-table and an op-code table is made. These tables are specific to the file (input) this operates on. If multiple files are involved, each file will have its' own table and stored in a file on its name.

PASS2 (ASSEMBLER): A symbol table and a variable table is made and appended to the file made at the end of the pass1. The variables and the symbols are replaced by the addresses. Initially, we assume the file loads at 0 and this value is updated in loader module.

LINKER: If the code is distributed over files, then the proper addressing for variables is done. Extern variables are handled.

LOADER: The user is asked for the memory location where he wants to load his program. The programs are then dynamically loaded into those specific memory locations. This particular thing was shown by displaying the programs in a different file, with lines depicting the address.

OUTPUT: There a number files generated for every input file given. ".pre", ".table", ".s", ".l.8085", ".s.8085".

FILE DESCTIPTION:

.pre : Consists of the entire code without the macros. This means the macros used in the code are replaced and that code is present in this file.

.table: This consists of the Macro-Table, Symbol-Table and Variable-Table.

.s: The variables and the symbols are replaced by their addresses

.l.8085: Link among various input files is done.

.s.8085: Final code with all the changes made and loaded at the place mentioned by the user.

INSTRUCTION SET:

It will work on all instructions defined in 8085 assembly code and the following instruction set as well.

ADDMM &m1,&m2,&m3 : Add 2 numbers stored in m2 and m3 memory locations. Stack starts from m1 memory location

ADDMR &m1,&m2 : Add number in the register to the number in m2 memory. Stack starts from m1 memory location

ADDVV &m1,&m2,v1,v2 : Add 2 variables v1 and v2 and store in m2 memory location. Stack starts from m1 memory location

ADDVR &m1,&m2,v1 : Add variable v1 to number stored in m2 memory location. Stack starts from m1 memory location

SUBMM &m1,&m2,&m3	: Subtract numbers stored in m3 from m2 memory location. Stack starts from m1 memory location
SUBMR &m1,&m2	: Subtract number in register from number in m2 memory location. Stack starts from m1 memory location
SUBVV &m1,&m2,v1,v2	: Subtract 2 variable v2 from v1 and store in m2 memory location. Stack starts from m1 memory location
SUBVR &m1,&m2,v1	: Subtract variable v1 from number stored in m2 memory location. Stack starts from m1 memory location
MINM &m1,&m2	: Find minimum between the numbers stored in 2 memory locations
MAXM &m1,&m2	: Find maximum between the numbers stored in 2 memory locations
MINMM &m1,&m2,&m3	: Find minimum among the numbers stored in 3 memory locations
MAXMM &m1,&m2,&m3	: Find maximum among the numbers stored in 3 memory locations
SWP &m1,&m2	: Swap the elements present in 2 memory locations
MUT &m1, &m2	: Multiply a number at memory location m2 with 2. Stack starts from memory location m1