CS-244
Assembler Linker Loader Documentation

for

System Programming Lab

Prepared by:
Group-12
Mitansh Jain - 160101042
Sujoy Ghosh - 160101073
Akul Agrawal - 160101085

Contents

List of Figures

List of Tables

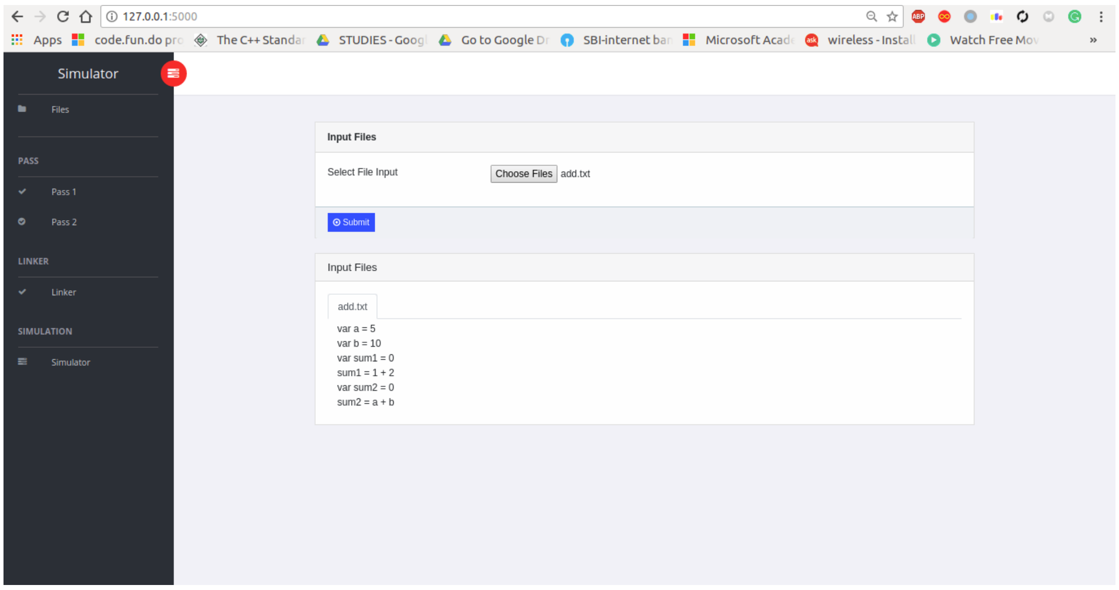
1 Introduction

This is a documnatation made for how to use the assembler, linker, loader web based simulartor and about its working.
Documentation consists mainly of following segemnts:

1. Firstly document covers on how to use the simulator and how it works.
2. Then it covers working of following:
   - Assembler
   - Linker
   - Loader
3. Also it covers various features of the c-type language implemented.

2 Simulator

2.1 File Upload



```
var a = 5
var b = 10
var sum1 = 0
sum1 = 1 + 2
var sum2 = 0
sum2 = a + b
```

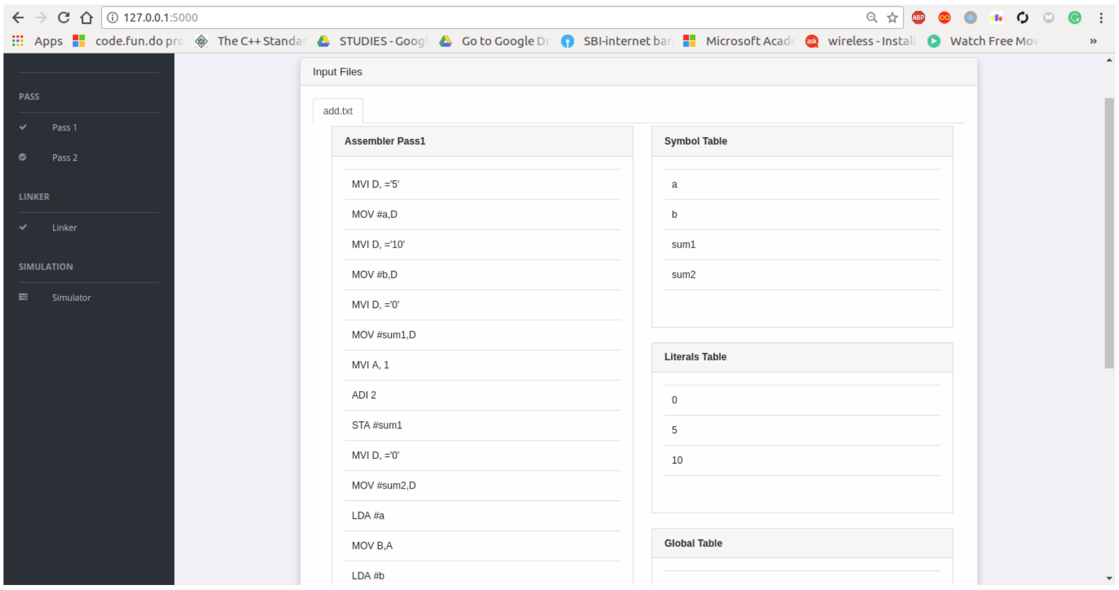
Figure 2.1: File upload window

This is the first screen of our webbased simuator, it is used to upload file to be simulated. To use it:

- Click on "Choose Files" button
- Then click submit button to upload files.
- After clicking on submit button contents of selected files will be visible below it.

After this to see output of assembler's pass1, pass2 or linker click on repective button on side nav bar. After clicking on them following screens show up.

2.2 Pass1 and Pass2



127.0.0.1:5000

Apps | code.fun.do pro | The C++ Standar | STUDIES - Googl | Go to Google Dr | SBI-internet bar | Microsoft Acad | wireless - Install | Watch Free Mov

PASS
- Pass 1
- Pass 2

LINKER
- Linker

SIMULATION
- Simulator

Input Files

add.txt

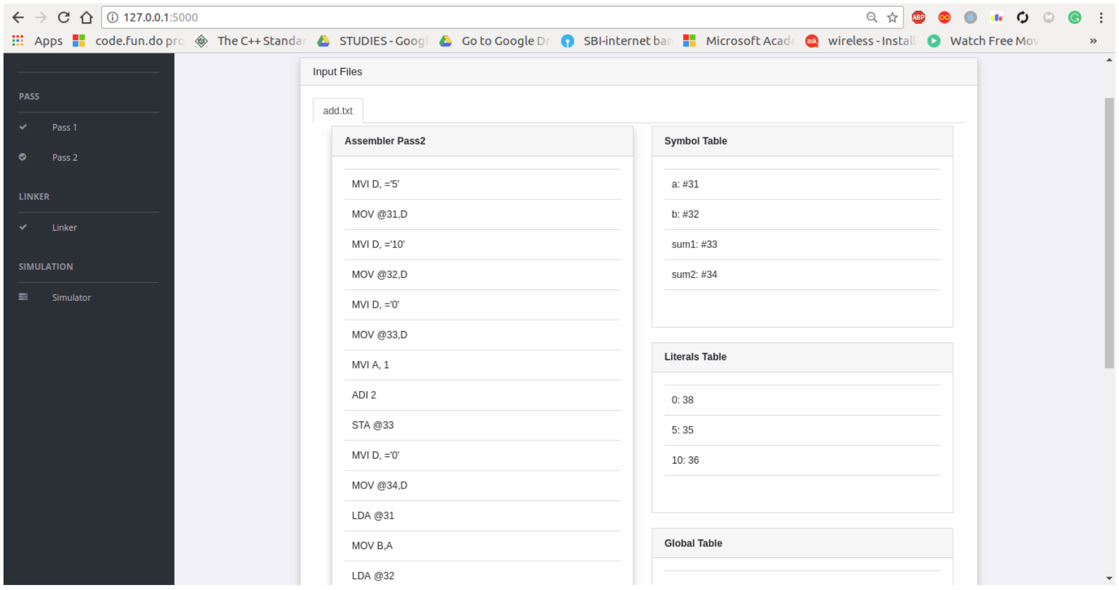
Assembler Pass1
MVI D, ='5'
MOV #a,D
MVI D, ='10'
MOV #b,D
MVI D, ='0'
MOV #sum1,D
MVI A, 1
ADI 2
STA #sum1
MVI D, ='0'
MOV #sum2,D
LDA #a
MOV B,A
LDA #b

Symbol Table
a
b
sum1
sum2

Literals Table
0
5
10

Global Table

Figure 2.2: Pass1 assembly, symbol table, literal table and global table



127.0.0.1:5000

Apps | code.fun.do pro | The C++ Standar | STUDIES - Googl | Go to Google Dr | SBI-internet bar | Microsoft Acad | wireless - Install | Watch Free Mov

PASS
- Pass 1
- Pass 2

LINKER
- Linker

SIMULATION
- Simulator

Input Files

add.txt

Assembler Pass2
MVI D, ='5'
MOV @31,D
MVI D, ='10'
MOV @32,D
MVI D, ='0'
MOV @33,D
MVI A, 1
ADI 2
STA @33
MVI D, ='0'
MOV @34,D
LDA @31
MOV B,A
LDA @32

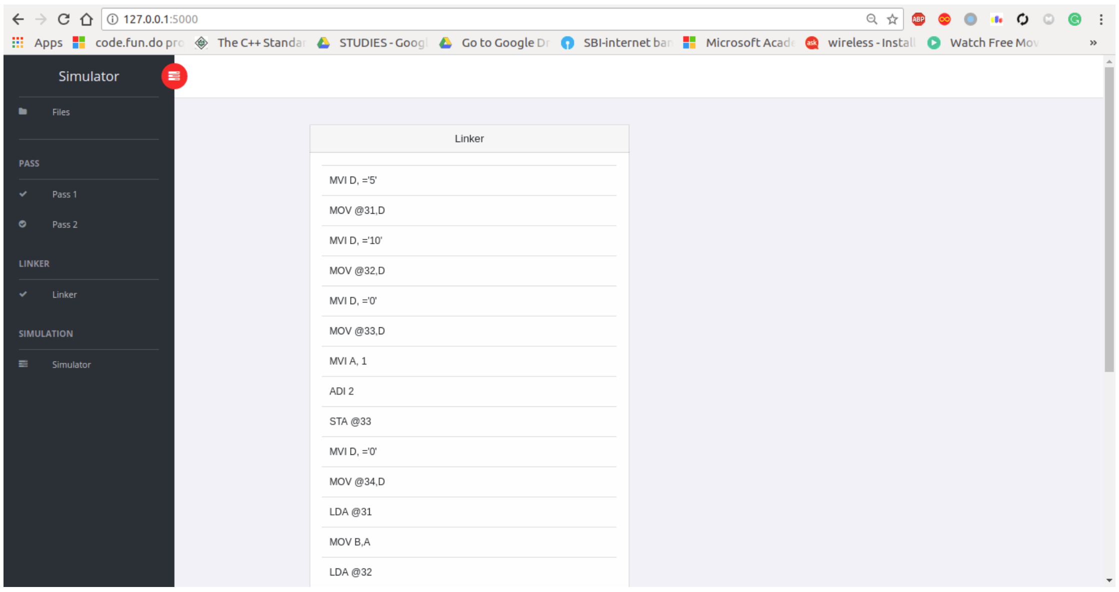
Symbol Table
a: #31
b: #32
sum1: #33
sum2: #34

Literals Table
0: 38
5: 35
10: 36

Global Table

Figure 2.3: Pass2 assembly, symbol table, literal table and global table

2.3 Linker

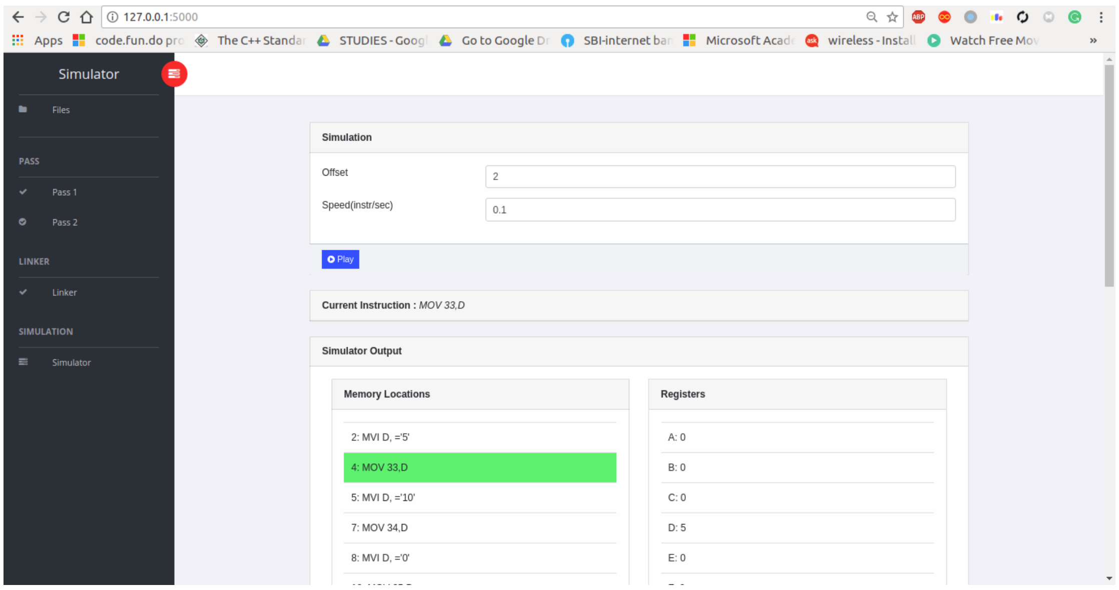


```
MVI D, ='5'
MOV @31,D
MVI D, ='10'
MOV @32,D
MVI D, ='0'
MOV @33,D
MVI A, 1
ADI 2
STA @33
MVI D, ='0'
MOV @34,D
LDA @31
MOV B,A
LDA @32
```

Figure 2.4: linker window

- This window shows final assembly code for the code uploaded in fileupload window after linking all the files uploaded.
- To simulate this code see the instruction on following page.

2.4 Simulator



Simulation

Offset: 2

Speed(instr/sec): 0.1

Current Instruction : *MOV 33,D*

Simulator Output

Memory Locations	Registers
2: MVI D, ='5'	A: 0
4: MOV 33,D	B: 0
5: MVI D, ='10'	C: 0
7: MOV 34,D	D: 5
8: MVI D, ='0'	E: 0

Figure 2.5: Linker window

- To simulate first enter value for offset and speed of execution.
- Next click on play button.
- You will see the code will start executing with line under execution highlighted with green.

3 Assembler

3.1 Data Structures Used

1. **Location Counter(LC)**
    - To implement memory allocation a data structure called location counter(LC) is introduced.
    - The location counter is always made to contain the address of the mnext memory word in the target program.
    - It is initialised to the constant specified in the START statment.
    - Whenever the analysis phase sees a label and the contents of LC in a new entry of the symbol table.
    - It then finds the number of memeory words required by the assembly statement and updates the LC contents.
2. **Operands Table - OPTAB:**
    - OPTAB contains the fields mnemonic opcode, class and mnemonic info.
    - The class field indicates whether the opcode corresponds to an imperative statemnts(IS), declaraive statements(DS) or an assembler directive.
    - Next it contains length of mnemonic info - its length and hexcode.
3. **Symbols Table - SYMTAB:**
    - A symbol table contains symbol, adress and length of various variables.
    - While processing of an assembly language it contains label, it is entered into symbol field of symtab and value of location counter is copied into length field.
    - The length of symbol is also entered into symtab if defined in statement.
    - For eg: consider the statement *a DS 4*
      Symbol: a
      Length: 4
      Adress: It will be value of location counter(LC)
4. **LITTAB and POOLTAB**
    - The first pass uses LITTAB to collect all literals used in program.

- Awareness of different literals pools is maintained using the auxillary table POOLTAB.
- POOLTAB contains the literal number of the starting itera of each literal pool.
- On encountoring LTORG or END statement all the literals are allocated adresses

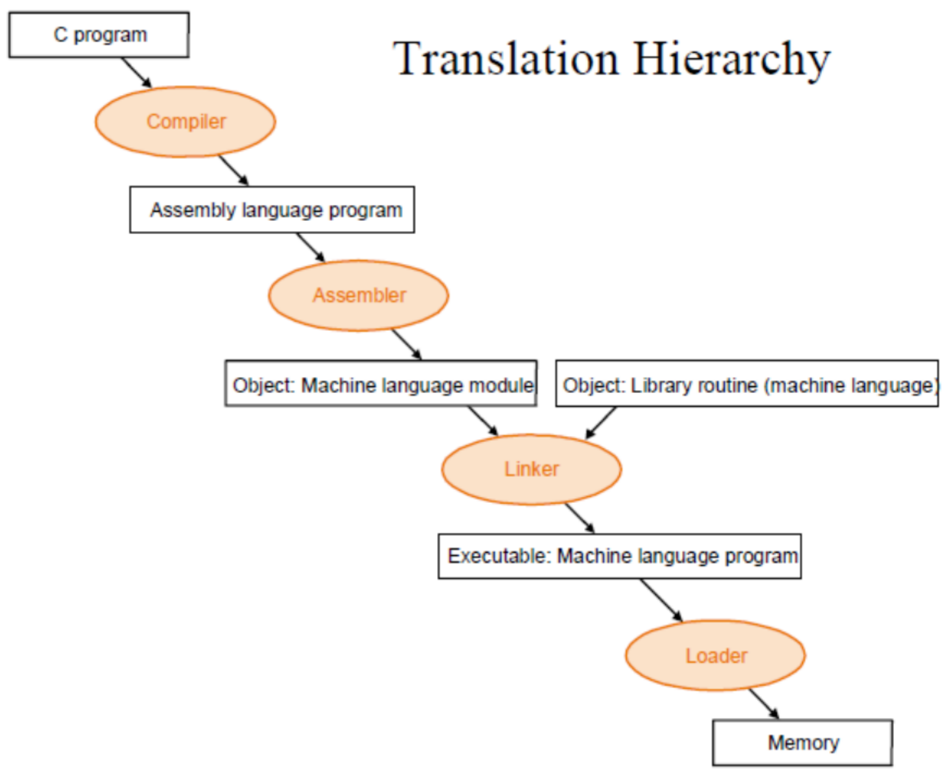

Translation Hierarchy
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Figure 3.1: Translation Hierachy

3.2 Pass1 of Assembler

3.2.1 Psuedocode for algorithm used

```
1. LC=0;(default values)

   pooltab_ptr=1;

   POOLTAB[1]=1;

   littab_ptr=1;
```

```
2. While next statement is not an END statement
    a) If label is pesent then
            this_label= symbol in the label field;
            Enter(this_label, LC) in SYMTAB;
    b) If START or ORIGIN statement then
            LC = value specified in operand field;
    c) If an EQU statement then
        i. this_addr = value of <adress_spec>;
        ii. Correct the symtab entry for this_label to(this_label, this_addr)
    d) If a declaration satement then
        i. code = code of the declaration statement;
        ii. size = size of memeory area required by DC/DS
        iii. LC = LC + size;
        iv. Generate(DL, code)...
    e) If an imperative statement then
        i. code = machine opcode from OPTAB
        ii. LC = LC + instruction length from OPTAB
        iii. If operand is a literal then
                this_literal = litera inoperand field;
                LITTAB[littab_ptr]=this_literal;
                littab_ptr = littab_ptr + 1;
        iv. else(i.e. operand is a symbol)
                this_entry = SYMTAB entry number of operand;
                Generate IC
3. (Processing of END statement)
    a) Process all literals to allocate memory and put the address field. Update LC
       accordingly.
    b) Generate IC'(AD, 02)'
    c) Go to Pass2
```

3.3 Pass1 of Assembler

3.3.1 Psuedocode for algorithm used

Algorithm 4.2 (Assembler Second Pass)

```
1. code_area_address = address of code_area;
   pooltab_ptr z= I;
   loc_cntr = 0;
2. While next statement is not an END statement
   a) Clear machine_code_buffer;
   b) If an LTORG statement
      i. Process literals in LIITTAB[POOLTAB[pooltab_ptr]]...LITTAB [POOLTAB
         [pooltab_ptr+1]]-1 similar to processing of constants in a DC statement,
         i.e. assemble the literals in machine_code_buffer.
      ii. size := size of memory area required for literals;
      iii. pooltab_ptr := pooltab_ptr + 1;
   c) If a START or ORIGIN statement then
      i. loc_cntr := value specified in operand field;
      ii. size := 0;
   d) If a declaration statement
      i. If a DC statement then Assemble the constant in machine_code_buffer.
      ii. size := size of memory area required by DC/DS;
   e) If an imperative statement
      i. Get operand address from SYMTAB or LITT AB.
      ii. Assemble instruction in machine_code_buffer.
      iii. size := size of instruction;
   f) If size $\neq$ 0 then
      i. Move contents of machine_code_buffer to the address code_area_address
         + loc_cntr;
      ii. loc_cntr := loc_cntr + size;
3. (Processing of END statement)
   a) Perform steps 2(b) and 2(f).
   b) Write code_area into output file.
```

4 Opcode Table of Intel 8085 Processor

Table 4.1: Opcode Table

First entry	Mnemonic	Fields Entry	Opcode	Length	Length
1.	ACI	Data	CE	2	
2.	ADC	A	8F	1	
3.	ADC	B	88	1	
4.	ADC	C	89	1	
5.	ADC	D	8A	1	
6.	ADC	E	8B	1	
7.	ADC	H	8C	1	
8.	ADC	L	8D	1	
9.	ADC	M	8E	1	
10.	ADD	A	87	1	
11.	ADD	B	80	1	
12.	ADD	C	81	1	
13.	ADD	D	82	1	
14.	ADD	E	83	1	
15.	ADD	H	84	1	
16.	ADD	L	85	1	
17.	ADD	M	86	1	
18.	ADI	Data	C6	2	
19.	ANA	A	A7	1	
20.	ANA	B	A0	1	
21.	ANA	C	A1	1	
22.	ANA	D	A2	1	
23.	ANA	E	A3	1	
24.	ANA	H	A4	1	
25.	ANA	L	A5	1	
26.	ANA	M	A6	1	
27.	ANI	Data	E6	2	
28.	CALL	Label	CD	3	
29.	CC	Label	DC	3	
30.	CM	Label	FC	3	
31.	CMA	2F	1		
32.	CMC	3F	1		
33.	CMP	A	BF	1	

Continued on next page

Table 4.1 – *Continued from previous page*

Ser. No.	**Mnemonic**	**Fields Entry**	**Opcode**		
34.	CMP	B	B8	1	
35.	CMP	C	B9	1	
36.	CMP	D	BA	1	
37.	CMP	E	BB	1	
38.	CMP	H	BC	1	
39.	CMP	L	BD	1	
40.	CMP	M	BD	1	
41.	CNC	Label	D4	3	
42.	CNZ	Label	C4	3	
43.	CP	Label	F4	3	
44.	CPE	Label	EC	3	
45.	CPI	Data	FE	2	
46.	CPO	Label	E4	3	
47.	CZ	Label	CC	3	
48.	DAA	27	1		
49.	DAD	B	09	1	
50.	DAD	D	19	1	
51.	DAD	H	29	1	
52.	DAD	SP	39	1	
53.	DCR	A	3D	1	
54.	DCR	B	05	1	
55.	DCR	C	0D	1	
56.	DCR	D	15	1	
57.	DCR	E	1D	1	
58.	DCR	H	25	1	
59.	DCR	L	2D	1	
60.	DCR	M	35	1	
61.	DCX	B	0B	1	
62.	DCX	D	1B	1	
63.	DCX	H	2B	1	
64.	DCX	SP	3B	1	
65.	DI	F3	1		
66.	EI	FB	1		
67.	HLT	76	1		
68.	IN	Port-address	DB	2	
69.	INR	A	3C	1	
70.	INR	B	04	1	
71.	INR	C	0C	1	
72.	INR	D	14	1	
73.	INR	E	1C	1	
74.	INR	H	24	1	

Continued on next page

Table 4.1 – *Continued from previous page*

Ser. No.	**Mnemonic**	**Fields Entry**	**Opcode**		
75.	INR	L	2C	1	
76.	INR	M	34	1	
77.	INX	B	03	1	
78.	INX	D	13	1	
79.	INX	H	23	1	
80.	INX	SP	33	1	
81.	JC	Label	DA	3	
82.	JM	Label	FA	3	
83.	JMP	Label	C3	3	
84.	JNC	Label	D2	3	
85.	JNZ	Label	C2	3	
86.	JP	Label	F2	3	
87.	JPE	Label	EA	3	
88.	JPO	Label	E2	3	
89.	JZ	Label	CA	3	
90.	LDA	Address	3A	3	
91.	LDAX	B	0A	1	
92.	LDAX	D	1A	1	
93.	LHLD	Address	2A	3	
94.	LXI	B	01	3	
95.	LXI	D	11	3	
96.	LXI	H	21	3	
97.	LXI	SP	31	3	
98.	MOV	A,	A	7F	1
99.	MOV	A,	B	78	1
100.	MOV	A,	C	79	1
101.	MOV	A,	D	7A	1
102.	MOV	A,	E	7B	1
103.	MOV	A,	H	7C	1
104.	MOV	A,	L	7D	1
105.	MOV	A,	M	7E	1
106.	MOV	B,	A	47	1
107.	MOV	B,	B	40	1
108.	MOV	B,	C	41	1
109.	MOV	B,	D	42	1
110.	MOV	B,	E	43	1
111.	MOV	B,	H	44	1
112.	MOV	B,	L	45	1
113.	MOV	B,	M	46	1
114.	MOV	C,	A	4F	1
115.	MOV	C,	B	48	1

Continued on next page

Table 4.1 – *Continued from previous page*

Ser. No.	Mnemonic	Fields Entry	Opcode		
116.	MOV	C,	C	49	1
117.	MOV	C,	D	4A	1
118.	MOV	C,	E	4B	1
119.	MOV	C,	H	4C	1
120.	MOV	C,	L	4D	1
121.	MOV	C,	M	4E	1
122.	MOV	D,	A	57	1
123.	MOV	D,	B	50	1
124.	MOV	D,	C	51	1
125.	MOV	D,	D	52	1
126.	MOV	D,	E	53	1
127.	MOV	D,	H	54	1
128.	MOV	D,	L	55	1
129.	MOV	D,	M	56	1
130.	MOV	E,	A	5F	1
131.	MOV	E,	B	58	1
132.	MOV	E,	C	59	1
133.	MOV	E,	D	5A	1
134.	MOV	E,	E	5B	1
135.	MOV	E,	H	5C	1
136.	MOV	E,	L	5D	1
137.	MOV	E,	M	5E	1
138.	MOV	H,	A	67	1
139.	MOV	H,	B	60	1
140.	MOV	H,	C	61	1
141.	MOV	H,	D	62	1
142.	MOV	H,	E	63	1
143.	MOV	H,	H	64	1
144.	MOV	H,	L	65	1
145.	MOV	H,	M	66	1
146.	MOV	L,	A	6F	1
147.	MOV	L,	B	68	1
148.	MOV	L,	C	69	1
149.	MOV	L,	D	6A	1
150.	MOV	L,	E	6B	1
151.	MOV	L,	H	6C	1
152.	MOV	L,	L	6D	1
153.	MOV	L,	M	6E	1
154.	MOV	M,	A	77	1
155.	MOV	M,	B	70	1
156.	MOV	M,	C	71	1

Continued on next page

Table 4.1 – *Continued from previous page*

Ser. No.	**Mnemonic**	**Fields Entry**	**Opcode**		
157.	MOV	M,	D	72	1
158.	MOV	M,	E	73	1
159.	MOV	M,	H	74	1
160.	MOV	M,	L	75	1
161.	MVI	A,	Data	3E	2
162.	MVI	B,	Data	06	2
163.	MVI	C,	Data	0E	2
164.	MVI	D,	Data	16	2
165.	MVI	E,	Data	1E	2
166.	MVI	H,	Data	26	2
167.	MVI	L,	Data	2E	2
168.	MVI	M,	Data	36	2
169.	NOP	00	1		
170.	ORA	A	B7	1	
171.	ORA	B	B0	1	
172.	ORA	C	B1	1	
173.	ORA	D	B2	1	
174.	ORA	E	B3	1	
175.	ORA	H	B4	1	
176.	ORA	L	B5	1	
177.	ORA	M	B6	1	
178.	ORI	Data	F6	2	
179.	OUT	Port-Address	D3	2	
180.	PCHL	E9	1		
181.	POP	B	C1	1	
182.	POP	D	D1	1	
183.	POP	H	E1	1	
184.	POP	PSW	F1	1	
185.	PUSH	B	C5	1	
186.	PUSH	D	D5	1	
187.	PUSH	H	E5	1	
188.	PUSH	PSW	F5	1	
189.	RAL	17	1		
190.	RAR	1F	1		
191.	RC	D8	1		
192.	RET	C9	1		
193.	RIM	20	1		
194.	RLC	07	1		
195.	RM	F8	1		
196.	RNC	D0	1		
197.	RNZ	C0	1		

Continued on next page

Table 4.1 – *Continued from previous page*

Ser. No.	Mnemonic	Fields Entry	Opcode		
198.	RP	F0	1		
199.	RPE	E8	1		
200.	RPO	E0	1		
201.	RRC	0F	1		
202.	RST	0	C7	1	
203.	RST	1	CF	1	
204.	RST	2	D7	1	
205.	RST	3	DF	1	
206.	RST	4	E7	1	
207.	RST	5	EF	1	
208.	RST	6	F7	1	
209.	RST	7	FF	1	
210.	RZ	C8	1		
211.	SBB	A	9F	1	
212.	SBB	B	98	1	
213.	SBB	C	99	1	
214.	SBB	D	9A	1	
215.	SBB	E	9B	1	
216.	SBB	H	9C	1	
217.	SBB	L	9D	1	
218.	SBB	M	9E	1	
219.	SBI	Data	DE	2	
220.	SHLD	Address	22	3	
221.	SIM	30	1		
222.	SPHL	F9	1		
223.	STA	Address	32	3	
224.	STAX	B	02	1	
225.	STAX	D	12	1	
226.	STC	37	1		
227.	SUB	A	97	1	
228.	SUB	B	90	1	
229.	SUB	C	91	1	
230.	SUB	D	92	1	
231.	SUB	E	93	1	
232.	SUB	H	94	1	
233.	SUB	L	95	1	
234.	SUB	M	96	1	
235.	SUI	Data	D6	2	
236.	XCHG	EB	1		
237.	XRA	A	AF	1	
238.	XRA	B	A8	1	

Continued on next page

Table 4.1 – *Continued from previous page*

Ser. No.	**Mnemonic**	**Fields Entry**	**Opcode**		
239.	XRA	C	A9	1	
240.	XRA	D	AA	1	
241.	XRA	E	AB	1	
242.	XRA	H	AC	1	
243.	XRA	L	AD	1	
244.	XRA	M	AE	1	
245.	XRI	Data	EE	2	
246.	XTHL	E3	1		

5 Linker

5.1 Introduction

It is a tool that mergees the object file produced by seperate compilation or asembly and creates an executable file. It does its task in three steps;

1. Searches the progren to find library routines used by program, e.g. print(), sqrt() and various others.

2. Determines the memory locations that code from each module wil occupy and relocales its instructions by adjsting absolute references

   Relocation: which modifies the object programs so that it can be loaded at an address different from the location orginally specified.

3. It combines two or more separate cores and supplies the information needed to allow references between them.

4. Computer programs typically comprise several parts or modules; all these parts/-modules need not be contained within a single object file, and in such case refer to each other by means of symbols. Typically, an object file can contain three kinds of symbols:

   - Publicly defined symbols, which allow it to be called by other modules , also called as public definition .
   - Externally defined symbols(undefined symbols), which calls the other modules where these symbols are defined, also called as external reference.
   - Local symbols, used internally within the object file to facilitate relocation.

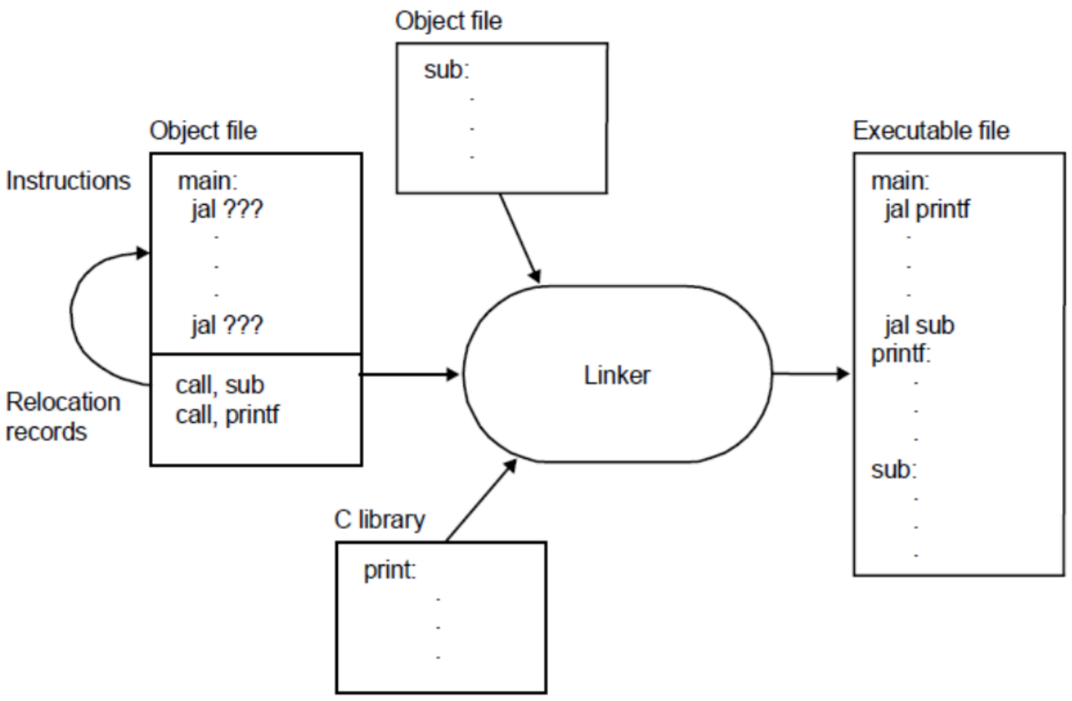


Figure 5.1: Basic Purpose of Linker Explained

5.2 Static Linking

1. Static linking is the result of the linker copying all library routines used in the program into the executable image.
2. This may require more disk space and memory than dynamic linking, but is more portable, since it does not require the presence of the library on the system where it runs.
3. Static linking also prevents ”DLL Hell”, since each program includes exactly the versions of library routines that it requires, with no conflict with other programs.
4. In addition, a program using just a few routines from a library does not require the entire library to be installed.

5.3 Dynamic Linking

1. Many operating system environments allow dynamic linking, that is the postponing of the resolving of some undefined symbols until a program is run.
2. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these.

3. Loading the program will load these objects/libraries as well, and perform a final linking. Dynamic linking needs no linker.

5.3.1 Advantages

1. Often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary.

2. If a bug in a library function is corrected by replacing the library, all programs using it dynamically will benefit from the correction after restarting them.

3. Programs that included this function by static linking would have to be re-linked first.

5.3.2 Disadvantages

1. Known on the Windows platform as "DLL Hell", an incompatible updated library will break executables that depended on the behavior of the previous version of the library if the newer version is not properly backward compatible.

2. A program, together with the libraries it uses, might be certified (e.g. as to correctness, documentation requirements, or performance) as a package, but not if components can be replaced.

5.4 Relocation

1. As the compiler has no information on the layout of objects in the final output, it cannot take advantage of shorter or more efficient instructions that place a requirement on the address of another object.

2. For example, a jump instruction can reference an absolute address or an offset from the current location, and the offset could be expressed with different lengths depending on the distance to the target.

3. By generating the most conservative instruction (usually the largest relative or absolute variant, depending on platform) and adding relaxation hints, it is possible to substitute shorter or more efficient instructions during the final link.

4. This step can be performed only after all input objects have been read and assigned temporary addresses; the linker relaxation pass subsequently reassigns addresses, which may in turn allow more relaxations to occur.

5. In general, the substituted sequences are shorter, which allows this process to always converge on the best solution given a fixed order of objects; if this is not the case, relaxations can conflict, and the linker needs to weigh the advantages of either option.

6 Loader

6.1 Introduction

It is a SYSTEM PROGRAM that brings an executable file residing on disk into memory and starts it running. Steps:-

1. Read executable file's header to determine the size of text and data segments.
2. Create a new address space for the program.
3. Copies instructions and data into address space.
4. Copies arguments passed to the program on the stack.
5. Initializes the machine registers including the stack pointer.
6. Jumps to a startup routine that copies the program's arguments from the stack to registers and calls the program's main routine.

There are mainly four types of loader. We have used a Direct Linking Loader. Description and their advantages and disadvantages are also mentioned.

6.2 Assemble-and-go Loader

1. Compilation, assembly, and link steps are not separated from program execution all in single pass.
2. The intermediate forms of the program are generally kept in RAM, and not saved to the file system.
3. Compile and go systems differ from interpreters, which either directly execute source code or execute an intermediate representation.

6.2.1 Advantages

1. The user need not be concerned with the separate steps of compilation, assembling, linking, loading, and executing.
2. Execution speed is generally much superior to interpreted systems. They are simple and easier to implement.

6.2.2 Disadvantages

1. There is wastage in memory space due to the presence of the assembler.
2. The code must be reprocessed every time it is run.
3. Systems with multiple modules, possibly in different languages, cannot be handled naturally within this framework.
4. Compile-and-go systems were popular in academic environments, where student programs were small, compiled many times, usually executed quickly and, once debugged, seldom needed to be re-executed.

6.3 Absolute Loader

It runs as absolute program.

6.3.1 Advantage:

1. Simple and efficient
2. No linking or relocation

6.3.2 Disadvantage:

1. Difficult to use subroutine libraries.
2. The need of programmer to state the actual address.

6.4 Bootstrap Loader

1. Special Type of Absolute Loader.
2. When a computer is first tuned on or restarted bootstrap loader is executed.
3. This bootstrap loads the first program to be run by computer that is the OS.
4. It loads the first address 0x80.

6.5 Direct Linking Loader

1. This type of loader is a relocating loader.
2. The loader cannot have the direct access to the source code.
3. To place the object code 2 types of addresses can be used:-

- ABSOLUTE : In this the absolute path of object code is known and the code is directly loaded in memory.
- RELATIVE :In this the relative path is known and this relative path is given by assembler.

6.5.1 Working

The assembler should give the following information to the loader:

1. The length of the object code segment.
2. A list of external symbols (could be used by diff. segment).
3. List of External symbols(The segment itself is using)
4. Information about address constants
5. Machine code translation of the source program

7 Language

This chpater enlists various featurs implemented in the language

7.1 Assignment and declaration

1. Variable Declaration

   Syntax:
   var variable_name

2. Array Declaration

   Syntax:
   var array_name[size]

3. Array Assignment

   Syntax:
   array_name[index] = value

7.2 Arithematic and Logical Operations

1. Addition

   Syntax:
   variable_name = variable_name/constant + variable_name/constant

2. Subtraction

   Syntax:
   variable_name = variable_name/constant - variable_name/constant

3. Division

   Syntax:
   variable_name = variable_name/constant / variable_name/constant

4. Multiplication

   Syntax:
   *variable_name = variable_name/constant * variable_name/constant*

5. Minimum

   Syntax:
   variable_name = min(list_of_variables(max 3))

6. Maximum

   Syntax:
   variable_name = max(list_of_variables(max 3))

7. And

   Syntax:
   variable_name = variable_name/constant and_symbol variable_name/constant

8. Or

   Syntax:
   variable_name = variable_name/constant or_symbol variable_name/constant

7.3 Loop

1. for loop

   Syntax:
   loop counter
   Code to be executed
   endloop

2. nested for loop

   Syntax:
   loop counter
   loop counter
   Code to be executed
   endloop
   endloop

7.4 If statemnt

1. If statement

   Syntax:
   If condition
   Code to be executed

endif

7.5 Global variables

1. global: used for declaring global variables

   Syntax:
   global variable_name

2. extern: used for using global variables

   Syntax:
   extern variable_name

7.6 Macros

1. Macro Function

   Syntax:
   macro
   function_name
   functin_body
   mend

2. Macro Variables

   Syntax:
   macro
   macro_variable_name
   mend

7.7 Functions

1. Function: used for declaring global variables

   Syntax:
   function function_name
   function_body
   endfunction

7.8 Goto Statements

1. label and jump: used for declaring global variables

   Syntax:
   jump: label_name
   code
   label_name: code_to_be_executed