

ASSEMBLERS

Prakash Patel- GIT

Design Specifications

- ❑ Identify the information necessary to perform a task
- ❑ Design a suitable data structure to record the information
- ❑ Determine the processing necessary to obtain and maintain the information
- ❑ Determine the processing necessary to perform the task

Synthesis Phase

□ MOVER BREG, ONE

- ▣ Address of the memory word with which name ONE is associated (depends on source program, so it must be made available by the analysis phase)
- ▣ Machine op codes corresponding to the mnemonic MOVER (not depends on source program, it depends on the assembly language)

□ Use two data structures:

- ▣ Symbol Table (name, address) – build by analysis phase
- ▣ Mnemonic Table (mnemonic, opcode, length)

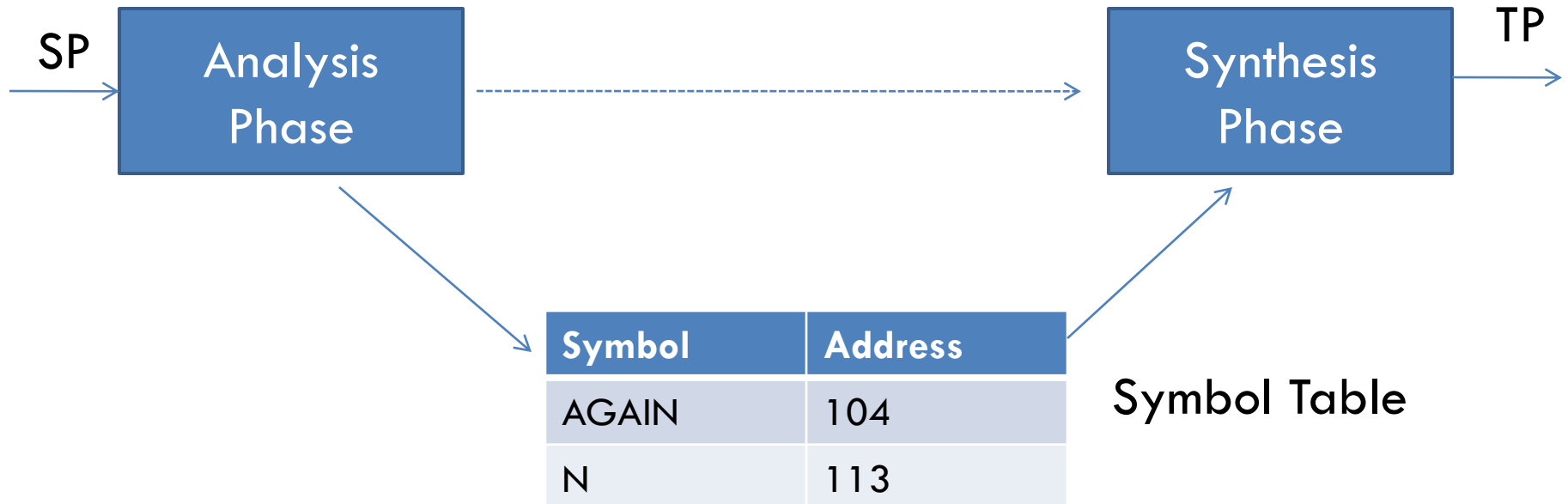
Analysis Phase

- ❑ The primary function is of building of the symbol table.
- ❑ Concept of “Memory Allocation”
- ❑ To implement memory allocation a data structure called location counter (LC) is used.
- ❑ The LC is always made to contain the address of the next memory word in the target program.
- ❑ It is initialized to the constant specified in the START statement.
- ❑ To update the contents of LC, analysis phase needs to know lengths of different instructions.

Data structures of the assembler

Mnemonic	Opcode	Length
ADD	01	1
SUB	02	1

Mnemonic Table

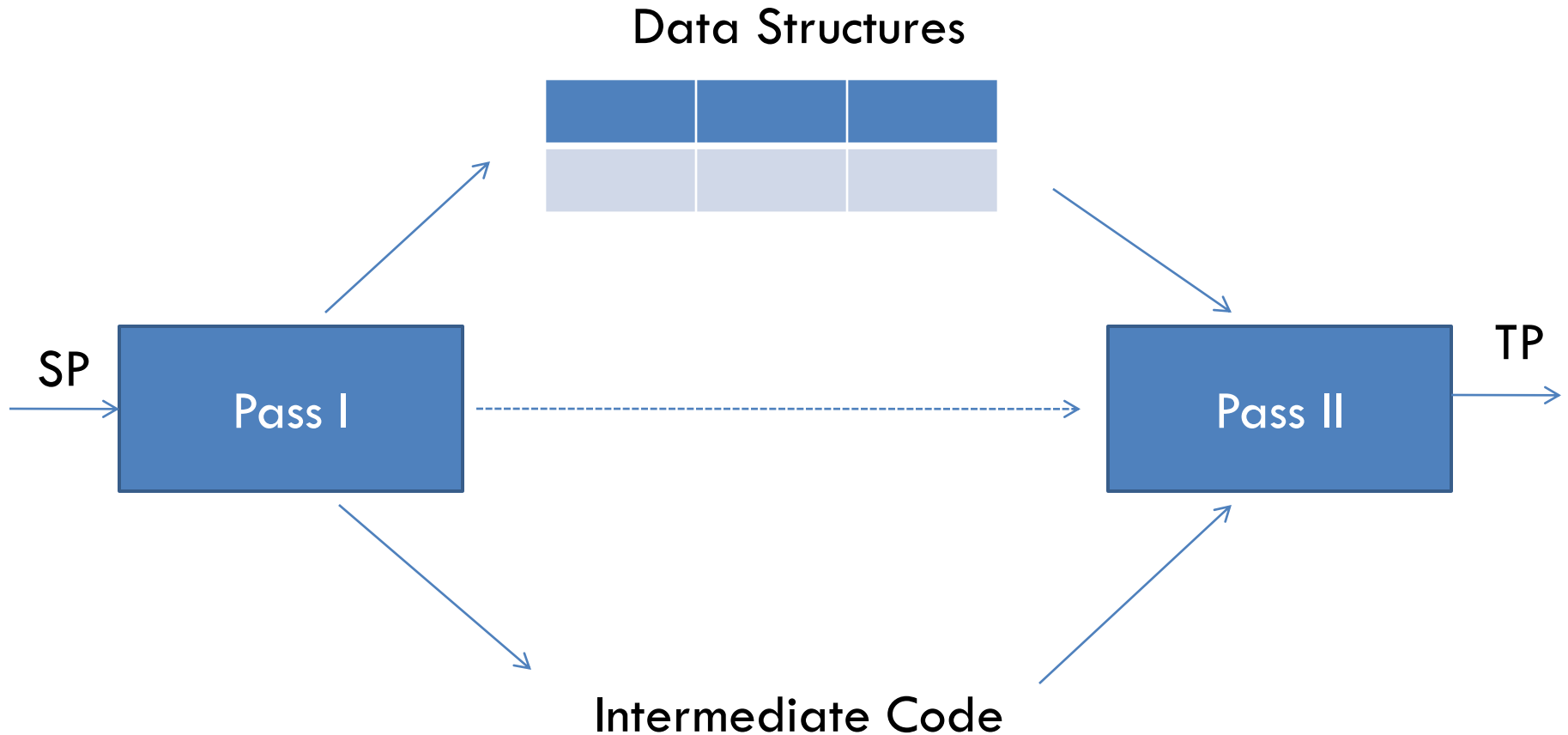


Pass Structure of Assembler

□ Two Pass Translation

- ▣ It can handle forward references easily.
- ▣ LC processing is performed in the first pass and symbol defined in the program are entered into the symbol table.
- ▣ The second pass synthesizes the target form using the address information found in the symbol table.
- ▣ In effect, the first pass performs analysis of the source program while the second pass performs synthesis of the target program.

Two pass assembly



Pass Structure of Assembler

- ❑ Single Pass Translation
 - ▣ LC processing and construction of the symbol table proceeds as in two pass translation.
 - ▣ The problem of forward references is tackled using a process called “backpatching”
 - ▣ The operand field of an instruction containing a forward reference is left blank initially. The address of the forward referenced symbol is put into this field when its definition is encountered.
 - ▣ `MOVER BREG, ONE` [ONE is forward reference]
 - ▣ Table of Incomplete Instruction (TII)

Advanced Assembler Directives

❑ ORIGIN

- ❑ ORIGIN <address spec>
- ❑ Where <address spec> is an <operand spec> or <constant>
- ❑ This directive indicates that LC should be set to the address given by <address spec>.
- ❑ It is useful when the target program does not consist of consecutive memory words.

Advanced Assembler Directives

□ EQU

- ▣ <symbol> EQU <address spec>
- ▣ Where <address spec> is an <operand spec> or <constant>
- ▣ It defines the symbol to represent <address spec>.

□ LTORG

- ▣ It permits a programmer to specify where literals should be placed.
- ▣ By default, assembler places it after the END statement

Literals

ADD	AREG,	= '5'	⇒		ADD	AREG,	FIVE
					--	--	
				FIVE	DC		'5'
		(a)				(b)	

- At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of a literal pool. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.

1		START	200		
2		MOVER	AREG, ='5'	200)	+04 1 211
3		MOVEM	AREG, A	201)	+05 1 217
4	LOOP	MOVER	AREG, A	202)	+04 1 217
5		MOVER	CREG, B	203)	+05 3 218
6		ADD	CREG, ='1'	204)	+01 3 212
7		...			
12		BC	ANY, NEXT	210)	+07 6 214
13		LTORG			
			= '5'	211)	+00 0 005
			= '1'	212)	+00 0 001
14		...			
15	NEXT	SUB	AREG, ='1'	214)	+02 1 219
16		BC	LT, BACK	215)	+07 1 202
17	LAST	STOP		216)	+00 0 000
18		ORIGIN	LOOP+2		
19		MULT	CREG, B	204)	+03 3 218
20		ORIGIN	LAST+1		
21	A	DS	1	217)	
22	BACK	EQU	LOOP		
23	B	DS	1	218)	
24		END			
25			= '1'	219)	+00 0 001

Design of a Two Pass Assembler

- ❑ Tasks performed by the passes of a two pass assembler are as follows:
 - ❑ Pass I
 - ▣ Separate the symbol, mnemonic opcode, operand fields
 - ▣ Build the symbol table
 - ▣ Perform LC processing
 - ▣ Construct intermediate representation
 - ❑ Pass II
 - ▣ Synthesis the target program

Pass I of an assembler

- It uses the following data structures:
 - ▣ OPTAB – A table of mnemonic opcodes and related information
 - ▣ SYMTAB – Symbol Table
 - ▣ LITTAB – A table of literals used in the program

Data Structures of assembler Pass I

<i>mnemonic opcode</i>	<i>class</i>	<i>mnemonic info</i>
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

OPTAB

<i>symbol</i>	<i>address</i>	<i>length</i>
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

SYMTAB

	<i>literal</i>	<i>address</i>
1	= '5'	
2	= '1'	
3	= '1'	

LITTAB

<i>literal no</i>
#1
#3
—

POOLTAB

Algorithm- Assembler First Pass

1. *loc_cntr* := 0; (default value)
pooltab_ptr := 1; POOLTAB[1] := 1;
littab_ptr := 1;
2. While next statement is not an END statement
 - (a) If label is present then
this_label := symbol in label field;
Enter (*this_label*, *loc_cntr*) in SYMTAB.
 - (b) If an LTORG statement then
 - (i) Process literals LITTAB[POOLTAB[*pooltab_ptr*]] ... LITTAB[*littab_ptr* - 1] to allocate memory and put the address in the *address* field. Update *loc_cntr* accordingly.
 - (ii) *pooltab_ptr* := *pooltab_ptr* + 1;
 - (iii) POOLTAB[*pooltab_ptr*] := *littab_ptr*;
 - (c) If a START or ORIGIN statement then
loc_cntr := value specified in operand field;
 - (d) If an EQU statement then
 - (i) *this_addr* := value of <*address spec*>;
 - (ii) Correct the symtab entry for *this_label* to (*this_label*, *this_addr*).

- (e) If a declaration statement then
 - (i) *code* := code of the declaration statement;
 - (ii) *size* := size of memory area required by DC/DS.
 - (iii) *loc_cntr* := *loc_cntr* + *size*;
 - (iv) Generate IC '(DL, *code*) ...'.
- (f) If an imperative statement then
 - (i) *code* := machine opcode from OPTAB;
 - (ii) *loc_cntr* := *loc_cntr* + instruction length from OPTAB;
 - (iii) If operand is a literal then
 - this_literal* := literal in operand field;
 - LITTAB [*littab_ptr*] := *this_literal*;
 - littab_ptr* := *littab_ptr* + 1;
 - else (i.e. operand is a symbol)
 - this_entry* := SYMTAB entry number of operand;
 - Generate IC '(IS, *code*)(S, *this_entry*)'

3. (Processing of END statement)

- (a) Perform step 2(b).
- (b) Generate IC '(AD,02)'.
- (c) Go to Pass II.

Intermediate code for Imperative Statements



We consider two variants of intermediate code which differ in the information contained in their operand fields. For simplicity, the address field is assumed to contain identical information in both variants.

Variant I and Variant II

Variant I

	START	200	(AD,01)	(C,200)
	READ	A	(IS,09)	(S,01)
LOOP	MOVER	AREG, A	(IS,04)	(1)(S,01)
	⋮		⋮	
	SUB	AREG, ='1'	(IS,02)	(1)(L,01)
	BC	GT, LOOP	(IS,07)	(4)(S,02)
	STOP		(IS,00)	
A	DS	1	(DL, 02)	(C,1)
	LTORG		(DL,05)	
	

The first operand is represented by a single digit number which is a code for a register (1-4 for AREG-DREG) or the condition code itself (1-6 for LT-ANY). The second operand, which is a memory operand, is represented by a pair of the form

(operand class, code)

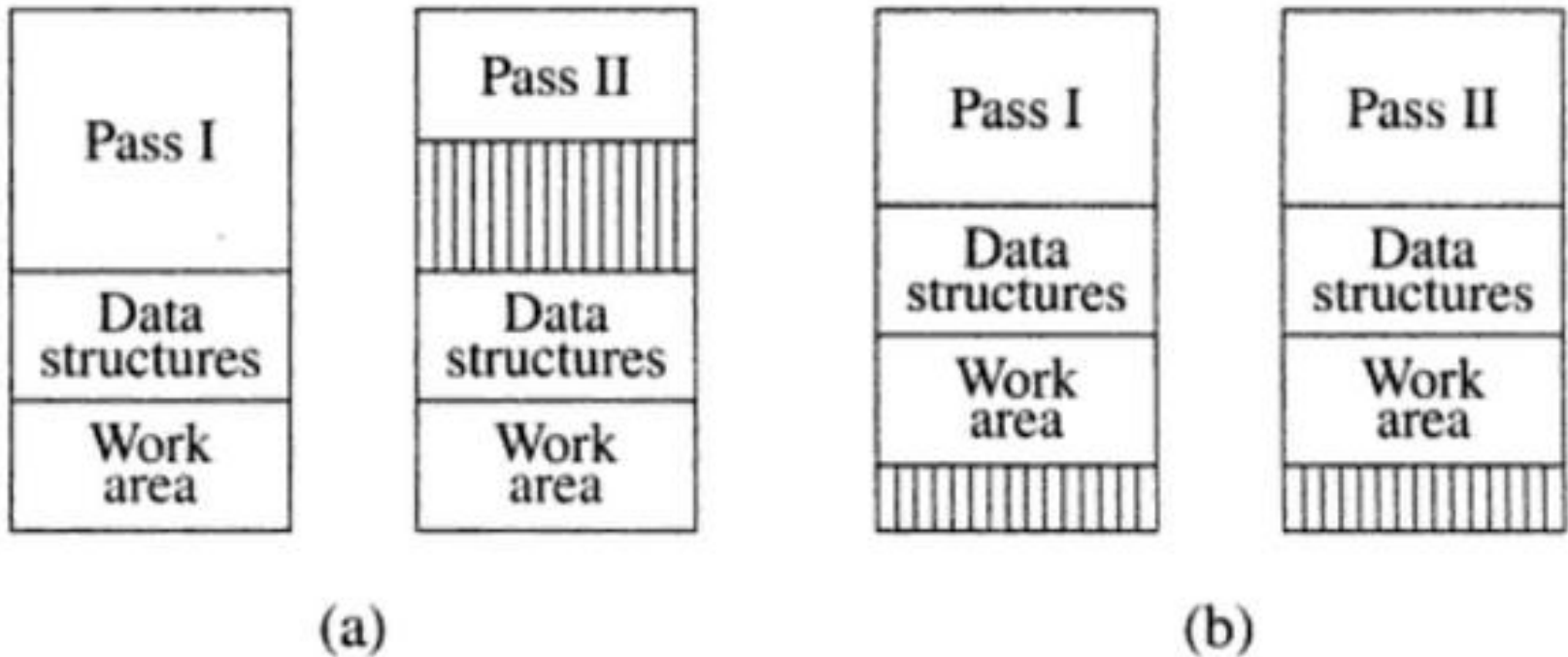
where *operand class* is one of C, S and L standing for constant, symbol and literal, respectively (see Fig. 4.12). For a constant, the *code* field contains the internal representation of the constant itself. For example, the operand descriptor for the statement `START 200` is (C, 200). For a symbol or literal, the *code* field contains the ordinal number of the operand's entry in SYMTAB or LITAB. Thus entries for a symbol XYZ and a literal ='25' would be of the form (S, 17) and (L, 35) respectively.

Variant II

This variant differs from variant I of the intermediate code in that the operand fields of the source statements are selectively replaced by their processed forms (see Fig. 4.13). For declarative statements and assembler directives, processing of the operand fields is essential to support LC processing. Hence these fields contain the processed forms. For imperative statements, the operand field is processed only to identify literal references. Literals are entered in LITTAB, and are represented as (L, *m*) in IC. Symbolic references in the source statement are not processed at all during Pass I.

	START	200	(AD,01)	(C,200)
	READ	A	(IS,09)	A
LOOP	MOVER	AREG, A	(IS,04)	AREG, A
	⋮		⋮	
	SUB	AREG, = '1'	(IS,02)	AREG, (L,01)
	BC	GT, LOOP	(IS,07)	GT, LOOP
	STOP		(IS,00)	
A	DS	1	(DL,02)	(C,1)
	LTORG		(DL,05)	
	---		---	

Comparison of the variants



Memory requirements using (a) variant I, (b) variant II

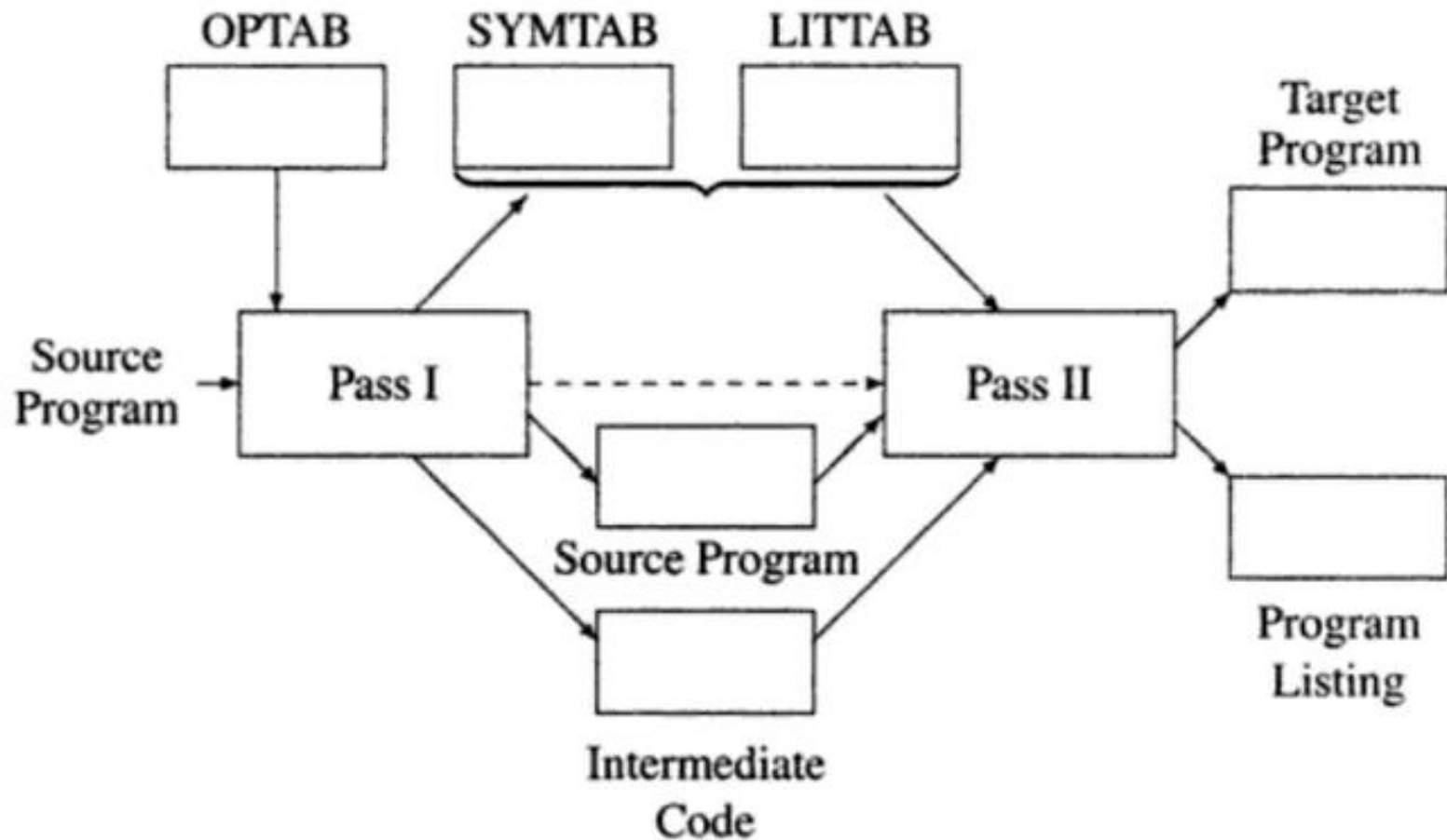
Comparison of the variants

Variant I of the intermediate code appears to require extra work in Pass I since operand fields are completely processed. However, this processing considerably simplifies the tasks of Pass II—a look at the IC of Fig. 4.12 confirms this. The functions of Pass II are quite trivial. To process the operand field of a declaration statement, we only need to refer to the appropriate table and obtain the operand address. Most declarations do not require any processing, e.g. DC, DS (see Section 4.4.5), and START statements, while some, e.g. LTORG, require marginal processing. The IC is quite compact—it can be as compact as the target code itself if each operand reference like (S, n) can be represented in the same number of bits as an operand address in a machine instruction.

Comparison of the variants

Variant II reduces the work of Pass I by transferring the burden of operand processing from Pass I to Pass II of the assembler. The IC is less compact since the memory operand of a typical imperative statement is in the source form itself. On the other hand, by making Pass II to perform more work, the functions and memory requirements of the two passes get better balanced. Figure 4.14 illustrates the advantages of this aspect. Part (a) of Fig. 4.14 shows memory utilization by an assembler using variant I of IC. Some data structures, viz. symbol table, are passed in the memory while IC is presumably written in a file. Since Pass I performs much more processing than Pass II, its code occupies more memory than the code of Pass II. Part

Pass II of an assembler



Pass II of an assembler

□ Tables

For efficiency reasons SYMTAB must remain in main memory throughout Passes I and II of the assembler. LITTAB is not accessed as frequently as SYMTAB, however it may be accessed sufficiently frequently to justify its presence in the memory. If memory is at a premium, it is possible to hold only part of LITTAB in the memory because only the literals of the current pool need to be accessible at any time. For obvious reasons, no such partitioning is feasible for SYMTAB. OPTAB should be in memory during Pass I.

Pass II of an assembler

□ Source Program and Intermediate Code

The source program would be read by Pass I on a statement by statement basis. After processing, a source statement can be written into a file for subsequent use in Pass II. The IC generated for it would also be written into another file. The target code and the program listings can be written out as separate files by Pass II. Since all these files are sequential in nature, it is beneficial to use appropriate blocking and buffering of records.