Two Pass Assembler

Read the source program twice

If in first pass all symbols are not found then signal error

In second pass generate the object code

Most assemblers are two pass.

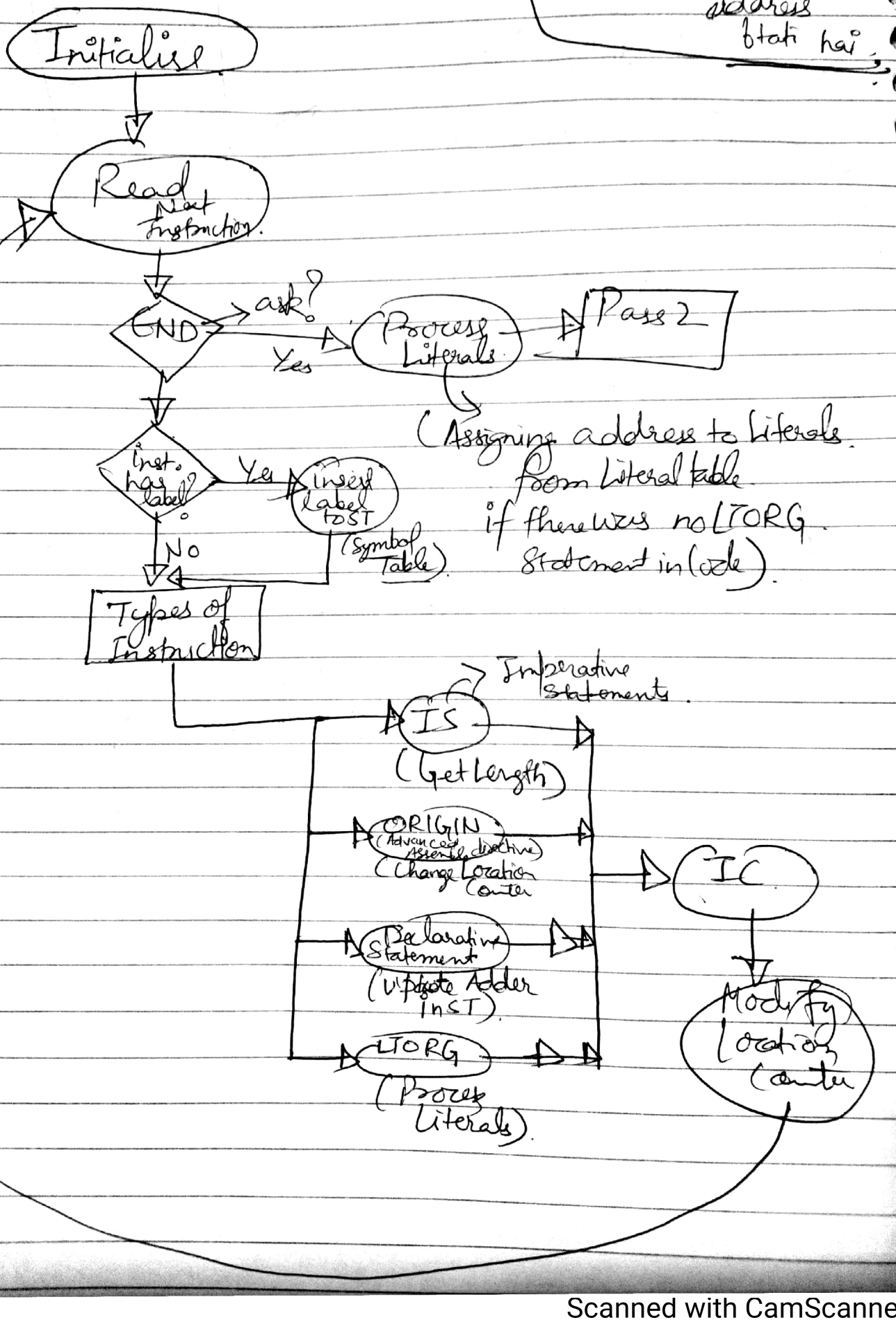
LTORG :- Is the baad wali statement se Literal table se address assign hone lgte hai

Pass1

Label → A sequence of characters that identifies a location within source code.

ORIGIN → जो भी इसके बाद Statement होती है उसका address बताती है।

Date.....



Date.....

Data Structures in Pass-1:

(I) Assembly Prog.

(II) Opcode table →

Mnemonic	Class	Machine Code	Length
STOP	IS	00	1
ADD	IS	01	1
SUB	IS	02	1
MUL	IS	03	1

(III) Symbol Table

(IV) Literal Table

Literal	address

After pass 1 Assembly code is converted to Intermediate code.

ST, LT and Pool table along with Intermediate Code is passed to pass 2.

Src Code	LC	IC
START 200		(AD,01) (C,200)
MOVER AREG, ='5'	200	(IS,04) (RG,01) (L,0)
MOVEM AREG, X	201	(IS,05) (RG,01) (S,0)
L_1 MOVER BREG, ='2'	202	(S,1) (IS,04) (RG,02) (L,1)
ORIGIN L_1+3	203	(AD,03) (C,205)
LTORG	205	(AD,05) (DL,02) (C,5)
	206	(AD,05) (DL,02) (C,2)
X DS 1	207	(S,0) (DL,01) (C,1)
END	208	(AD,02)

ST

	S	A
0	X	207
1	L_1	202

LT

	L	A
0	='5'	205
1	='2'	206

Pool

0

Algorithm for first Pass of the Assembler follows

```
public static void pass_one( ) {
  // This procedure is an outline of pass one of a simple assembler.
  boolean more_input = true;                    // flag that stops pass one
  String line, symbol, literal, opcode;         // fields of the instruction
  int location_counter, length, value, type;    // misc. variables
  final int END_STATEMENT = -2;                 // signals end of input

  location_counter = 0;                         // assemble first instruction at 0
  initialize_tables( );                         // general initialization

  while (more_input) {                          // more_input set to false by END
    line = read_next_line( );                   // get a line of input
    length = 0;                                 // # bytes in the instruction
    type = 0;                                   // which type (format) is the instruction

    if (line_is_not_comment(line)) {
      symbol = check_for_symbol(line);          // is this line labeled?
      if (symbol != null)                       // if it is, record symbol and value
        enter_new_symbol(symbol, location_counter);
      literal = check_for_literal(line);        // does line contain a literal?
      if (literal != null)                      // if it does, enter it in table
        enter_new_literal(literal);
```

Algorithm for Pass one continues.....

```
        // Now determine the opcode type. -1 means illegal opcode.
        opcode = extract_opcode(line);          // locate opcode mnemonic
        type = search_opcode_table(opcode); // find format, e.g. OP REG1,REG2
        if (type < 0)                                   // if not an opcode, is it a pseudoinstruction?
           type = search_pseudo_table(opcode);
        switch(type) {                                  // determine the length of this instruction
           case 1: length = get_length_of_type1(line);  break;
           case 2: length = get_length_of_type2(line);  break;
           // other cases here
        }
     }

     write_temp_file(type, opcode, length, line);   // useful info for pass two
     location_counter = location_counter + length;      // update loc_ctr
     if (type == END_STATEMENT) {           // are we done with input?
        more_input = false;                 // if so, perform housekeeping tasks
        rewind_temp_for_pass_two( );        // like rewinding the temp file
        sort_literal_table( );              // and sorting the literal table
        remove_redundant_literals( );       // and removing duplicates from it
     }
  }
}
```

Pass Two

The function of pass two is to generate the object program

The algorithm follows:

```
public static void pass_two( ) {
   // This procedure is an outline of pass two of a simple assembler.
   boolean more_input = true;                  // flag that stops pass two
   String line, opcode;                        // fields of the instruction
   int location_counter, length, type;         // misc. variables
   final int END_STATEMENT = -2;               // signals end of input
   final int MAX_CODE = 16;                    // max bytes of code per instruction
   byte code[ ] = new byte[MAX_CODE];          // holds generated code per instruction

   location_counter = 0;                       // assemble first instruction at 0

   while (more_input) {                        // more_input set to false by END
      type = read_type( );                     // get type field of next line
      opcode = read_opcode( );                 // get opcode field of next line
      length = read_length( );                 // get length field of next line
      line = read_line( );                     // get the actual line of input
```

Algorithm for Pass two continues.....

```
        if (type != 0) {                                    // type 0 is for comment lines
            switch(type) {                                  // generate the output code
                case 1: eval_type1(opcode, length, line, code);  break;
                case 2: eval_type2(opcode, length, line, code);  break;
                // other cases here
            }
        }
        write_output(code);                                 // write the binary code
        write_listing(code, line);                          // print one line on the listing
        location_counter = location_counter + length;       // update loc_ctr
        if (type == END_STATEMENT) {                        // are we done with input?
            more_input = false;                             // if so, perform housekeeping tasks
            finish_up( );                                   // odds and ends
        }
    }
}
```

Error Reporting

Some Common Errors

- A symbol has been used but not defined
- A symbol has been defined more than once
- The name in the opcode field is not a legal opcode
- An opcode is not supplied with enough operands
- An opcode is supplied with too many operands
- The END statement is missing

Assembler can print the error line and the error message and continue with assembly

Recovery upon error

- Skip to next instruction
- Cascading errors: absence of entries in tables lead to spurious error reporting