

National Institute Of Technology Goa

One Pass Assembler



Gautam Mishra 17CSE1011

Archit 17CSE1029

Supervisor: Dr. Veena Thenkanidiyoor

System Programming Assignment

Department of Computer Science and Engineering

April 18, 2019

Contents

1	Introduction	1
1.1	Problem Statement	1
2	How to Compile	2
3	Design & Data Structures	6
3.1	Design	6
3.2	Data Structure	7
4	Algorithm	8
5	References	10

Chapter 1

Introduction

An assembler is a translator that translates an assembler program into a conventional machine language program. Basically, the assembler goes through the program one line at a time, and generates machine code for that instruction. Then the assembler proceeds to the next instruction. In this way, the entire machine code program is created.

In one-pass assemblers, it generates the object code in memory for immediate execution. No object program is written out, and no loader is needed. This kind of load-and-go assembler is useful in a system that is oriented toward program development and testing. Because programs are re-assembled nearly every time they are run, the efficiency of the assembly process is an important consideration. A load-and-go assembler avoids the overhead of writing the object program out and reading it back in. However, a one-pass assembler also avoids the overhead of an additional pass over the source program. Because the object program is produced in memory rather than being written out on secondary storage, the handling of forward references becomes less difficult.

1.1 Problem Statement

The one pass assembler needs to scan the program only once and create the equivalent object code. The biggest challenge is to tackle the forward referencing problem. We face this problem when either the label or variable in operand is not declared before that statement. The assembler simply generates object code instructions as it scans the source program. If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled. The symbol used as an operand is entered into the symbol table (unless such an entry is already present). This entry is flagged to indicate that the symbol is undefined. The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry. When the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if one exists), and the proper address is inserted into any instructions previously generated.

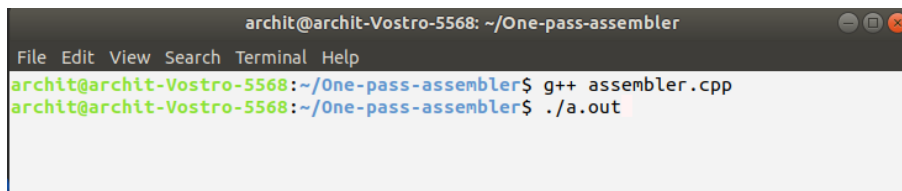
Chapter 2

How to Compile

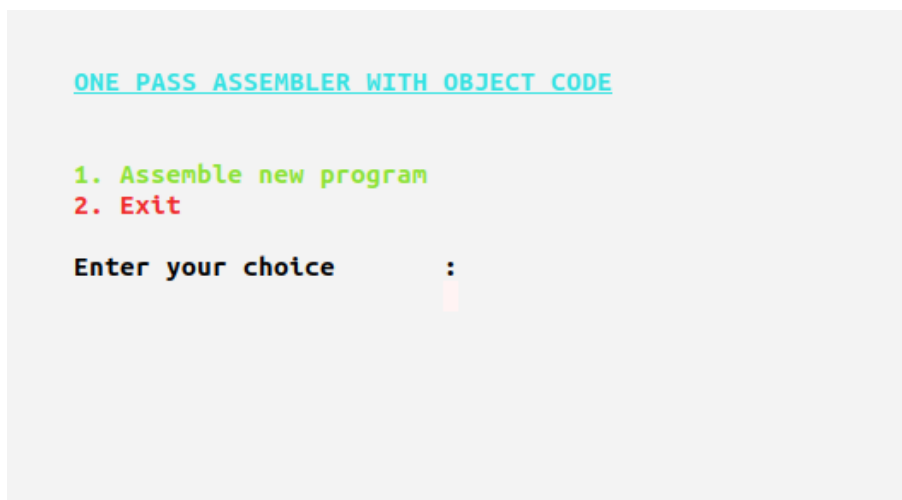
It is required to implement the One Pass assembler that generates object code written in C++ code.

Steps to Compile and execute the program:

1. In folder One-pass-assembler, open the terminal and run following commands:
 - (a) `g++ assembler.cpp`
 - (b) `./a.out`



```
archit@archit-Vostro-5568: ~/One-pass-assembler
File Edit View Search Terminal Help
archit@archit-Vostro-5568:~/One-pass-assembler$ g++ assembler.cpp
archit@archit-Vostro-5568:~/One-pass-assembler$ ./a.out
```



```
ONE PASS ASSEMBLER WITH OBJECT CODE

1. Assemble new program
2. Exit

Enter your choice :
```

2. Enter you choice, source file name and file name where Object code will be stored

```
ONE PASS ASSEMBLER WITH OBJECT CODE

1. Assemble new program
2. Exit

Enter your choice      :
                        1

Source File Name : InputSourcePgm.txt
File name where Object Code will be stored : ObjectCode.txt
```

3. After entering above details. Select your choice in next window to display : Source code, OPTAB, object code

```
ONE PASS ASSEMBLER WITH OBJECT CODE

1. Display source code
2. Display OPTAB
3. Display object code
4. Return to Main

Enter your choice      :
```

4. For Source Code:

```

=====
COPY      START    1000
EOF       BYTE     C'EOF'
THREE     WORD     3
ZERO      WORD     0
RETADR    RESW     1
LENGTH    RESW     1
BUFFER    RESB     4096
FIRST     STL      RETADR
CLOOP     JSUB     RDREC
          LDA      LENGTH
          COMP     ZERO
          JEQ      ENDFIL
          JSUB     WRREC
          J        CLOOP
ENDFIL    LDA      EOF
          STA      BUFFER
          LDA      THREE
          STA      LENGTH
          JSUB     WRREC
          LDL      RETADR
          RSUB

.
.        SUBROUTINE TO READ RECORD INTO BUFFER
.
INPUT     BYTE     X'F1'
MAXLEN    WORD     4096
RDREC     LDX      ZERO
.
          LDA      ZERO
RLOOP     TD       INPUT
          JEQ      RLOOP
          RD       INPUT
          COMP     ZERO
          JEQ      EXIT
          STCH     BUFFER,X
          TIX      MAXLEN
          JLT      RLOOP
EXIT      STX      LENGTH
          RSUB

.
.        SUBROUTINE TO WRITE RECORD FROM BUFFER
.
OUTPUT    BYTE     X'05'
WRREC     LDX      ZERO
WLOOP     TD       OUTPUT
          JEQ      WLOOP
          LDCH     BUFFER,X
          WD       OUTPUT
          TIX      LENGTH
          JLT      WLOOP
          RSUB
          END      FIRST
=====

```

5. Object Code :

```

ONE PASS ASSEMBLER WITH OBJECT CODE

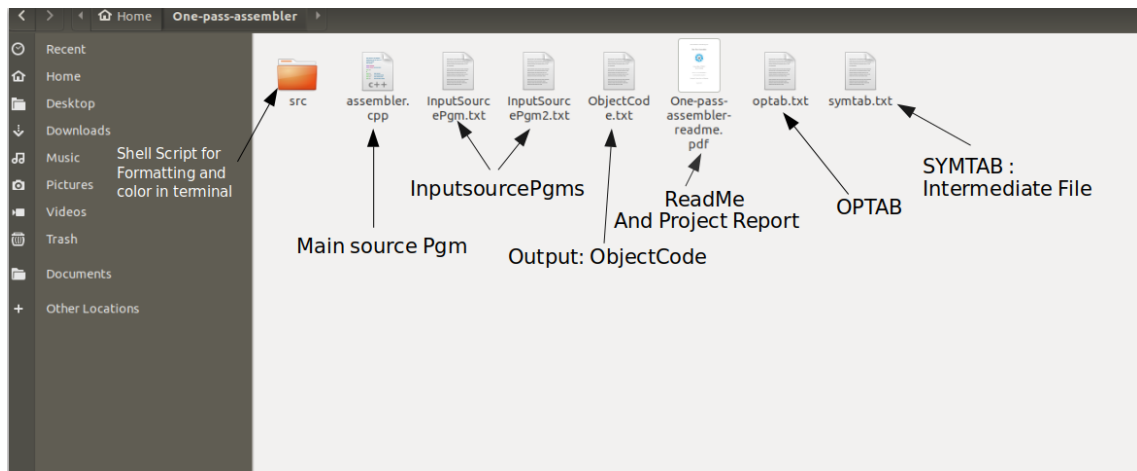
1. Display source code
2. Display OPTAB
3. Display object code
4. Return to Main

Enter your choice      :
                        3

=====
H^COPY  001000^00107A
T^001000^09^454F4F^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
=====

```

6. .zip file view



Chapter 3

Design & Data Structures

3.1 Design

The Design is consisting of 5 main modules(Control Unit-Parser-Address Convertors-OpTab-SymTab)

1. Control Unit:

- (a) Reading and storing the InputSourceFile and sending it to parser and store tokens in vector.
- (b) Assigning Address using LOCCTR.
- (c) Storing Symbol Table, Records and Opcode in map (Hash).
- (d) Write the symbol table and object code in file.

2. Parser:

This Module is responsible of parsing an instruction line into tokens(Labels, Opcode, Operands).

3. Address Convertors:

They are responsible for converting hexToDec, hexToBin and decToHex addresses.

4. OpTab:

It store opcodes in map

5. SymTab:

It store symbols in map and while execution symbols are stored in this table and whenever required search the symbols and also used to validate the program.

3.2 Data Structure

In this assembler various Data Structure used like Map, List and Vector to implement OPTAB, SYMTAB, Records, LOCCTR

Following Data Structure are implemented as given below:

OPTAB

1. OPTAB implemented as $\text{map} < \text{string}, \text{string} >$
2. In this key and value is stored as string.

SYMTAB

1. SYMTAB implemented as $\text{mapmap} < \text{string}, \text{pair} < \text{int}, \text{list} < \text{int} >>>$
2. In this key is stored as string.
3. In $\text{pair} < \text{int}, \text{list} < \text{int} >>$ LOCCTR(address) value of symbol and address of undefined symbol is added into list of forward references associated with symbol table entry.

Records

1. Records implemented as $\text{map} < \text{int}, \text{pair} < \text{int}, \text{vector} < \text{string} >>>$
2. In this it stores text record entries of object code.
3. In key it store record number and in $\text{vector} < \text{string} >$ it stores object codes.

LOCCTR

1. LOOCTR is implemented as integer and containing Decimal value of Locations(Hexadecimal).

Chapter 4

Algorithm

One-pass assemblers that produce object programs follow a slightly different procedure. Forward references are entered into lists as before. Now, however, when the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification. In general, they will already have been written out as part of a Text record in the object program. In this case the assembler must generate another Text record with the correct operand address. When the program is loaded, this address will be inserted into the instruction by the action of the loader

Algorithm to implement this assembler is as follows:

```
begin
  read first input line {from intermediate file}
  if OPCODE = 'START' then
    begin
      write listing line
      read next input line
    end {if START}
  write Header record to object program
  initialize first Text record
```

cont'd

```

while opcode != 'End' do
begin
    if there is no comment line then
    begin
        if there is a symbol in the LABEL field then
        begin
            search SYMTAB for LABEL
            if found then
            begin
                if <symbol value> as null
                set <symbol value> as LOCCTR and search
                    the linked list with corresponding
                    operand
                PTR addresses and generate operand
                    addresses as corresponding symbol
                    values
                set symbol value as LOCCTR in symbol table
                    and delete the linked list
            end
            else
                insert (LABEL,LOCCTR) into symtab
        end
        search OPTAB for OPCODE
        if found then
        begin
            search SYMTAB for OPERAND addresses
            if found then
                if symbol value not equal to null then
                    store symbol value as OPERAND address
                else
                    insert at the end of the linked list
                        with a node with address as LOCCTR
            else
                insert (symbol name,null)
                LOCCTR+=3
            end
            else if OPCODE='WORD' then
                add 3 to LOCCTR and convert comment to object code
            else if OPCODE='RESW' then
                add 3 #[OPERAND] to LOCCTR
            else if OPCODE='RESB' then
                add #[OPERAND] to LOCCTR
            else if OPCODE='Byte' then
            begin
                find the length of constant in bytes
                add length to LOCCTR
                convert constant to object code
            end
            if object code will not fit into current text record then
            begin
                write text record to object program initialize new Text record
            end
            add object code to Text record
        end
        write listing line
        read next input line
    end
    write last Text record to object program
    write End record to object program
    write last listing line

```

Chapter 5

References

Leland L. Beck, D. Manjula, System software: An introduction to systems programming, Pearson education, 3 rd ed, 2007.