Fr. Conceicao Rodrigues College of Engineering
Department of Computer Engineering

Academic Term : Jan-May 2024 - 25

Class : T.E. (Computer - A)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	7
Title:	program to implement 2 pass assembler
Date of Performance:	01/04/2025
Date of Submission:	16/04/2025
Roll No:	9913
Name of the Student:	Mark Lopes

Evaluation:

Sr. No	Rubric	Grade
1	Time Line (2)	
2	Output(3)	
3	Code optimization (2)	
4	Postlab (3)	

Signature of the Teacher :

Experiment No 7

Aim: Write a program to implement Two Pass Assembler.

Learning Objective: Translating mnemonic operation codes to their machine language equivalents. Assigning machine addresses to symbolic labels used by the programmer. Lastly to convert assembly language to binary.

Algorithm:

Pass 1:

1. Start

2.Intialize location counter to zero

3.Read opcode field of next instruction.

4.search opcode in pseudo opcode Table(POT)

5.If opcode is found in POT

5.1 If it is 'DS' or 'DC'

Adjust location counter to proper alignment.

Assign length of data field to 'L'

Go to step 9

5.2 If it is 'EQU'

Evaluate operand field

Assign values to symbol in label field

Go to step 3

5.3 If it is 'USING' or 'DROP' Go to step 3

5.4 If it is 'END'

Assign value to symbol in label field

Go to step 3

6. Search opcode in Machine Opcode Table.

7. Assign its length to ‘L’.

8. Process any literals and enter them into literal Table.

9.If symbol is there in the label field

Assign current value of Location Counter to symbol

10. Location Counter= Location Counter +L.

11.Go to step 3.

12. Stop.

Pass2:

1.Start

2.Intialize location counter to zero.

3. Read opcode field of next instruction.

4.Search opcode in pseudo opcode table.

5.If opcode is found in pseudo opcode Table

5.1 If it is ‘DS’ or ‘DC’

Adjust location counter to proper alignment.

If it is ‘DC’ opcode form constant and insert in assembled program

Assign length of data field to ‘L’

Go to step 6.4

5.2 If it is ‘EQU’ or ‘START’ ignore it. Go to step 3

5.3 If it is ‘USING’

Evaluate operand and enter base reg no. and value into base table

Go to step 3

5.4 If it is ‘DROP’

Indicate base reg no . available in base table . Go to step 3

5.5 If it is ‘END’

Generate literals for entries in Literal Table

Go to step 12

6 Search opcode in MOT

7. Get opcode byte and format code

8. Assign its length to ‘L’.

9. Check type of instruction.

10.If it is type ‘RR’ type

10.1 Evaluate both register expressions and insert into second byte.

10.2 Assemble instruction

10.3 Location Counter= Location Counter +L.

10.4.Go to step 3.

11. If it is ‘RX’ type

11.1 Evaluate register and index expressions and insert into second byte.

11.2 Calculate effective address of operand.

11.3 Determine appropriate displacement and base register

11.4 Put base and displacement into bytes 3 and 4

11.5 Location Counter= Location Counter +L.

11.6 Go to step 11.2

13 Stop.

Implementation Details

1. Read Assembly language input file.
2. Display output of Pass1 as the output file with Op-code Table, Symbol Table.
3. Display output of pass2 as the Op-code Table, Symbol Table , Copy file.

Test Cases:

1 Input symbol which is not defined

2 Input Opcode which is not entered in MOT

Conclusion:

Post Lab Questions:

1. **Define the basic functions of assembler.**

   Translation (Assembly to Machine Code)

   It translates the human-written assembly code into 0s and 1s (binary code) that the computer's hardware can actually run.

   Assigning Addresses (Location)

   It decides where each instruction and variable will be stored in memory, kind of like labeling where everything goes in a warehouse.

   Symbol Table Creation

   It keeps a list of all labels and variables in the program and their memory locations. Think of this as a map of important names and where they are kept.

   Handling Constants and Variables

   It takes care of values used in the program and ensures they are correctly placed in memory.

   Checking for Errors

   It checks the code for mistakes, like missing labels or invalid instructions, and lets the programmer know.

   Generating Object Code

   It creates the final "object code", which is the machine code stored in a file ready to be loaded and run by the computer.

2. **What is the need of SYMTAB (symbol table) in assembler?**

   SYMTAB is needed in an assembler to store the names of labels and variables along with their memory addresses. It helps the assembler keep track of where each symbol is located in memory. This is useful for translating instructions, handling forward references, and avoiding errors like duplicate or undefined symbols.

3. **What is the need of MOT in assembler**

   MOT is used by the assembler to store information about each machine instruction. It contains details like the instruction's name, its binary opcode, length, and format. When the assembler reads an instruction like L or A, it checks the MOT to find the correct opcode and how to translate it into machine language. So, MOT helps the assembler convert mnemonics into actual machine code.

4. **What is meant by one pass assembler?**

   A one-pass assembler is an assembler that reads the source code only once to translate it into machine code. It does everything like assigning addresses, building the symbol table, and generating object code in a single scan. It's faster and requires less memory, but it may not handle forward references (symbols used before they are defined) easily.

CODE:-

```
def process_assembly_code(code):
    # Clean up input - remove extra asterisks and split into
lines
    lines = [line.strip().replace('*', '') for line in
code.split('\n')]
    lines = [line for line in lines if line]  # Remove empty
lines

    # First pass: build symbol table and assign addresses
    symbol_table = {}
    location_counter = 0
    pass1_result = []

    for line in lines:
        parts = line.split()
        if not parts:  # Skip empty lines
            continue

        # Handle different line formats (with or without label)
        if parts[0] in ['START', 'USING', 'END']:
            # These are directives without labels
            label = ""
            operation = parts[0]
            operands = ' '.join(parts[1:]) if len(parts) > 1 else
""
            size = 0
        else:
```

```
            # Check if first part is a label or operation
            if parts[0] in ['L', 'A', 'ST', 'DC', 'DS']:
                # No label
                label = ""
                operation = parts[0]
                operands = ' '.join(parts[1:])
            else:
                # Has label
                label = parts[0]
                operation = parts[1]
                operands = ' '.join(parts[2:]) if len(parts) > 2
else ""


        # Determine size based on operation
        if operation in ['L', 'A', 'ST']:
            size = 4
        elif operation == 'DC':
            if 'H' in operands:
                size = 2
            elif 'F' in operands:
                size = 4
            else:
                size = 0
        elif operation == 'DS':
            if 'H' in operands:
                size = 2
            elif 'F' in operands or operands.isdigit() or
operands.startswith('1'):
```

```
                size = 4
            else:
                size = 0
        else:
            size = 0

        # Store symbol in symbol table if present
        if label and operation != 'END':
            symbol_table[label] = {'address': location_counter,
'size': size}  # Store address and size

        # Set pass1 and pass2 columns based on operation
        if operation == 'L':
            pass1_text = f"L, {operands.split(',')[0]}, -"
            if ',' in operands:
                operand_name = operands.split(',')[1]
                if operand_name in symbol_table:
                    addr = symbol_table[operand_name]['address']
                    pass2_text = f"06, {operands.split(',')[0]},
{addr}"
                else:
                    pass2_text = f"06, {operands.split(',')[0]},
??"
            else:
                pass2_text = "??"
        elif operation == 'A':
            pass1_text = f"A, {operands.split(',')[0]}, -"
            if ',' in operands:
```

```
                operand_name = operands.split(',')[1]
                if operand_name in symbol_table:
                    addr = symbol_table[operand_name]['address']
                    pass2_text = f"01, {operands.split(',')[0]},
{addr}"
                else:
                    pass2_text = f"01, {operands.split(',')[0]},
??"
            else:
                pass2_text = "??"
        elif operation == 'ST':
            pass1_text = f"ST, {operands.split(',')[0]}, -"
            if ',' in operands:
                operand_name = operands.split(',')[1]
                if operand_name in symbol_table:
                    addr = symbol_table[operand_name]['address']
                    pass2_text = f"05, {operands.split(',')[0]},
{addr}"
                else:
                    pass2_text = f"05, {operands.split(',')[0]},
??"
            else:
                pass2_text = "??"
        elif operation == 'DC':
            pass1_text = "-"
            if "'" in operands:
                value = operands.split("'")[1]
                pass1_text = f"{value}"
```

```
                pass2_text = f"{location_counter + size},{value}"
            else:
                pass2_text = f"_"
        elif operation == 'DS':
            pass1_text = "-"
            pass2_text = f"_"
        elif operation == 'USING':
            pass1_text = "-"
            pass2_text = "-"
        elif operation == 'START':
            pass1_text = "-"
            pass2_text = "-"
        elif operation == 'END':
            pass1_text = "-"
            pass2_text = f"_"
        else:
            pass1_text = "-"
            pass2_text = "-"

        # Record this line's information
        pass1_result.append({
            'address': location_counter,
            'label': label,
            'operation': operation,
            'operands': operands,
            'size': size,
            'pass1': pass1_text,
```

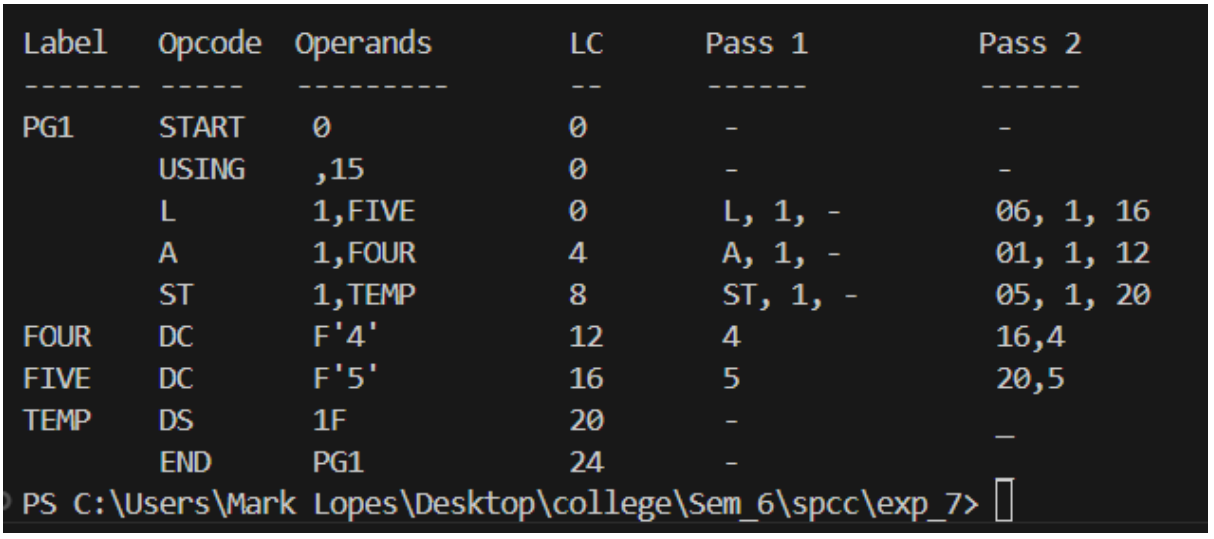
```
            'pass2': pass2_text
        })

        # Update location counter
        location_counter += size

    # Second pass: resolve symbols
    pass2_result = []

    for entry in pass1_result:
        # If it's a reference to a symbol, look it up in the symbol table
        if entry['operation'] in ['L', 'A', 'ST']:
            if ',' in entry['operands']:
                operand_name = entry['operands'].split(',')[1]
                if operand_name in symbol_table:
                    addr = symbol_table[operand_name]['address']
                    entry['pass2'] = entry['pass2'].replace('??', str(addr))

        pass2_result.append(entry)

    return symbol_table, pass1_result, pass2_result

def format_output(symbol_table, pass1_result, pass2_result):
    print("SYMBOL TABLE:")
    print("Symbol\tAddress\tSize")
    print("------\t-------\t----")
```

```
    for symbol, symbol_info in symbol_table.items():

print(f"{symbol}\t{symbol_info['address']}\t{symbol_info['size']}
")
    print()

    print("OUTPUT:")
    print("Label\tOpcode\tOperands\tLC\tPass 1\t\tPass 2")
    print("-------\t-----\t---------\t--\t------\t\t------")
    for entry in pass2_result:
        print(f"{entry['label']:<7} {entry['operation']:<8}
{entry['operands']:<14} {entry['address']:<8}
{entry['pass1']:<15} {entry['pass2']}")



def main():
    with open('input1.txt', 'r') as file:
        assembly_code = file.read()
    try:
        symbol_table, pass1_result, pass2_result =
process_assembly_code(assembly_code)

        format_output(symbol_table, pass1_result, pass2_result)

    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    main()
```

OUTPUT:-

input1.txt

```
PG1 START 0

** USING *,15

** L 1,FIVE

** A 1,FOUR

** ST 1,TEMP

FOUR DC F'4'

FIVE DC F'5'

TEMP DS 1F

** END PG1
```



```
Label   Opcode  Operands        LC      Pass 1          Pass 2
------- -----   ---------       --      ------          ------
PG1     START    0              0        -               -
        USING    ,15            0        -               -
        L        1,FIVE         0        L, 1, -         06, 1, 16
        A        1,FOUR         4        A, 1, -         01, 1, 12
        ST       1,TEMP         8        ST, 1, -        05, 1, 20
FOUR    DC       F'4'           12       4               16,4
FIVE    DC       F'5'           16       5               20,5
TEMP    DS       1F             20       -               _
        END      PG1            24       -
PS C:\Users\Mark Lopes\Desktop\college\Sem_6\spcc\exp_7>
```

input2.txt

```
PG1 START 0

** USING *,15

** L 1,FIVE

** A 1,FOUR

** ST 1,TEMP
```

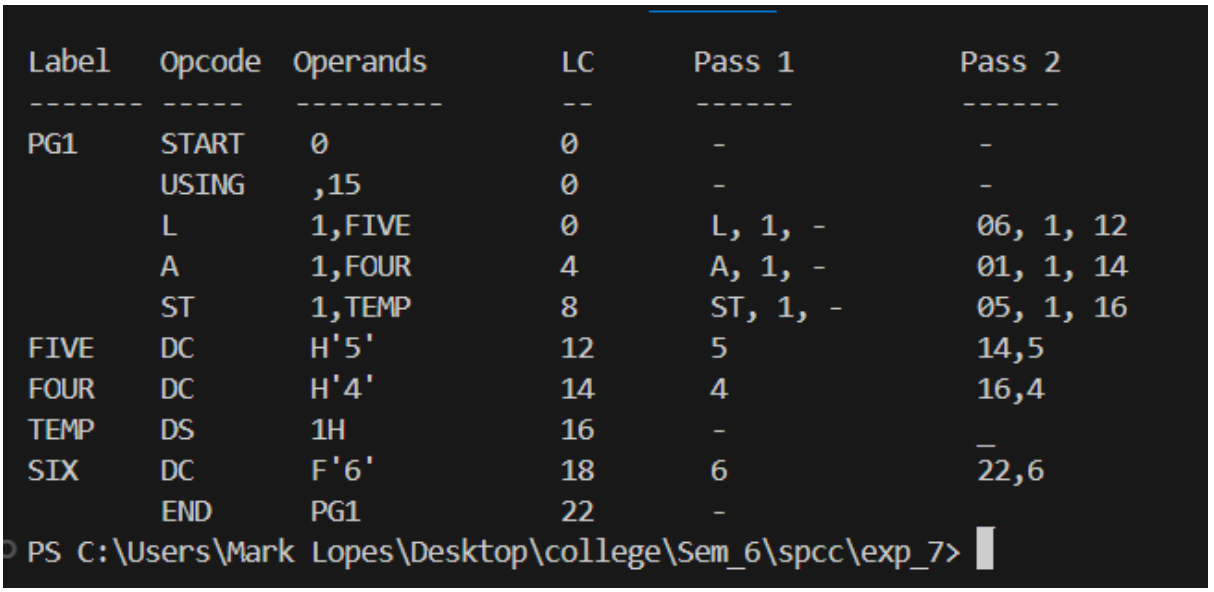
FIVE DC H'5'

FOUR DC H'4'

TEMP DS 1H

SIX DC F'6'

** END PG1



Label	Opcode	Operands	LC	Pass 1	Pass 2
PG1	START	0	0	-	-
	USING	,15	0	-	-
	L	1,FIVE	0	L, 1, -	06, 1, 12
	A	1,FOUR	4	A, 1, -	01, 1, 14
	ST	1,TEMP	8	ST, 1, -	05, 1, 16
FIVE	DC	H'5'	12	5	14,5
FOUR	DC	H'4'	14	4	16,4
TEMP	DS	1H	16	-	_
SIX	DC	F'6'	18	6	22,6
	END	PG1	22	-	

PS C:\Users\Mark Lopes\Desktop\college\Sem_6\spcc\exp_7>