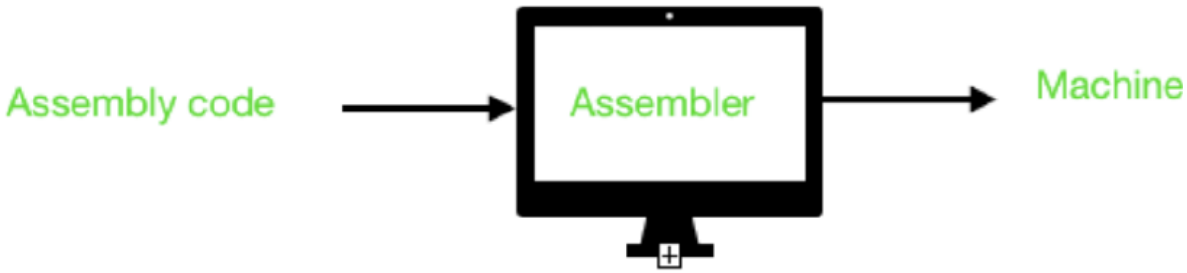
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300101
Name	Adwait Purao
Class and Batch	TE Computer Engineering - Batch B
Date	19/04/2024
Lab #	08
Aim	To design a two pass assembler.
Objective	To implement a Two-Pass Assembler: Develop and execute a program capable of converting assembly language instructions into machine code using a two-pass approach. To generate machine code: Execute Pass-2 of the assembler to convert symbolic opcodes into numeric opcodes, thereby generating machine code for the assembly language program.
Theory	Assembler: An assembler is a program that converts low-level assembly code into relocatable machine code and provides information for the loader. It's essential for translating user-written programs into machine code, which is the language computers understand. This translation from high-level language to machine language is facilitated by system software. Essentially, an assembler translates assembly language programs into machine language programs. A self-assembler is a program that operates on a computer and produces machine codes for that same computer or machine. It's also referred to as a resident assembler. On the other hand, a cross-assembler runs on one computer but generates machine codes for other computers. Operation Process: The assembler generates instructions by interpreting mnemonics (symbols) in the operation field and determining the values of symbols and literals to produce machine code. If an assembler completes all tasks in one scan, it's called a single-pass assembler; otherwise, it's referred to as a multiple-pass assembler. Assemblers typically divide these tasks into two passes:



Pass-1:

1. Define symbols and literals and store them in the symbol table and literal table respectively.
2. Keep track of the location counter.
3. Process pseudo-operations.
4. Define a program that assigns memory addresses to the variables and translates the source code into machine code.

Pass-2:

1. Generate object code by converting symbolic opcodes into their respective numeric opcodes.
2. Generate data for literals and find values of symbols.
3. Define a program that reads the source code two times.
4. Read the source code and translate it into object code.

```
[Label] [Opcode] [operand]

Example:  M  ADD  R1, ='3'
where, M - Label; ADD - symbolic opcode;
R1 - symbolic register operand; (='3') - Literal

Assembly Program:
Label  Op-code   operand   LC value(Location counter)
JOHN   START     200
       MOVER     R1, ='3'    200
       MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
       LTORG                 203
X      DS        1           204
       END                   205
```

Let's delve into how this program operates:

Execution Flow:

- **START:** This instruction initiates program execution from location 200, and the label "START" assigns a name to the program (named "JOHN").
- **MOVER:** It transfers the content of the literal (= '3') into the register operand R1.
- **MOVEM:** It moves the content of the register into the memory operand (X).
- **MOVER:** Once again, it transfers the content of the literal (= '2') into the register operand R2, with the label specified as L1.

- **LTORG:** This command assigns addresses to literals (current LC value).
- **DS (Data Space):** It allocates a data space of 1 to the symbol X.
- **END:** Marks the end of the program execution.

Working of Pass-1: At this stage, we define the symbol and literal tables along with their addresses. Note: Literal address is determined by LTORG or END.

1. **START 200:** Since no symbols or literals are found, both tables remain empty.
2. **MOVER R1, ='3' 200:** As '3' is a literal, the literal table is created.
3. **MOVEM R1, X 201:** X is a symbol referred to prior to its declaration, so it's stored in the symbol table with a blank address field.
4. **L1 MOVER R2, ='2' 202:** Both L1 (label) and '2' (literal) are stored in their respective tables.
5. **LTORG 203:** The address is assigned to the first literal specified by LC value, i.e., 203.
6. **X DS 1 204:** Although X is assigned data space of 1, it's referred to earlier, resulting in a Forward Reference Problem. The assembler assigns X the address specified by the LC value of the current step.
7. **END 205:** Marks the program's end, and the remaining literal receives the address specified by the LC value of the END instruction.

Pass-2: In this phase, the assembler generates machine code by converting symbolic machine opcodes into their respective bit configurations. It stores all machine opcodes in the MOT table (opcode table) with their symbolic code, length, and bit configuration. Additionally, it processes pseudo-ops and stores them in the POT table (pseudo-op table). Various databases are required for Pass-2, including the MOT table, POT table, Base table (storing the value of the base register), and LC (location counter).

Implementation / Code

```
class DualPhaseAssembler:
    def __init__(self):
        self.symbol_map = {}

    def initial_traverse(self, code_lines):
        pos_counter = 0
        for entry in code_lines:
            segments = entry.split()
            if len(segments) == 0:
                continue
            if segments[0] not in ['START', 'END']:
                if segments[0] not in self.symbol_map:
                    self.symbol_map[segments[0]] = pos_counter
            if segments[0] == 'START':
                pos_counter = int(segments[1], 16)
            elif segments[0] == 'END':
                break
            else:
                pos_counter += 1

    def secondary_traverse(self, code_lines):
        machine_language = []
        for entry in code_lines:
            segments = entry.split()
            if len(segments) == 0:
                continue
            if segments[0] == 'END':
                break
            if segments[0] in ['START', 'END']:
                continue
            if len(segments) > 1:
                if segments[1] in self.symbol_map:
                    machine_language.append(hex(self.symbol_map[segments[1]]
)[2:])
                else:
                    machine_language.append(segments[1])
        return machine_language

    def assemble_code(self, code_lines):
        self.initial_traverse(code_lines)
        return self.secondary_traverse(code_lines)
```

```python
# Usage example:
source_code = [
    "START 1000",
    "LOOP LDA VAL",
    "STA SUM",
    "INX",
    "BNE LOOP",
    "HLT",
    "VAL DC 05",
    "SUM DS 01",
    "END"
]

assembler = DualPhaseAssembler()
machine_code = assembler.assemble_code(source_code)
print("Machine Code:", machine_code)
```

Output

```
PS D:\adwait_8> python main.py
Machine Code: ['LDA', '1006', '1000', 'DC', 'DS']
PS D:\adwait_8>
```

Conclusion

Our experiment involved constructing a two-step assembler using Python to translate assembly language instructions into machine code. During the initial phase, we established symbol and numerical representations, managed positional information, and addressed special commands. Subsequently, in the second phase, we converted symbols into numerical equivalents, resolved symbol values, and generated the final machine code. Through this project, we gained valuable insights into the intricacies of assembler functionality, including symbol management and code generation. Ultimately, our assembler effectively fulfilled its purpose by seamlessly transforming assembly code into machine code, a critical component in the process of program development.

References

1. OpenAI. (2022). ChatGPT [Computer software]. Retrieved from https://openai.com/chatgpt
2. Slideshare. (n.d.). Design of Two Pass Assembler in System Software. Retrieved from https://www.slideshare.net/slideshow/10design-of-two-pass-assembler-in-system-softwarepdf/266918869
3. GeeksforGeeks. (n.d.). Single Pass, Two Pass, and Multi Pass Compilers. Retrieved from https://www.geeksforgeeks.org/single-pass-two-pass-and-multi-pass-compilers/