I will be updating this file. Please keep checking for the updates.

Assemblers:

The Assembler is Software that converts an assembly language code to machine code. It takes basic Computer commands and converts them into Binary Code that Computer's Processor can use to perform its Basic Operations. These instructions are assembler language or assembly language which is simpler and more human-readable than the language computers use.

We can also name an assembler as the compiler of assembly language. This is because a compiler converts the high-level language to machine language.

Elements of Assembly Language Programming-

Assembly language programming is a way to give instructions to a computer's central processing unit (CPU) using a language that's more human-readable than machine code but still closely related to it.

1. Instructions/ Command-A command is an instruction in assembly code that tells the assembler what action to take. Assembly language commands frequently use abbreviations to keep the terminology short while also using self-descriptive abbreviations, such as "ADD" for addition and "MOV" for data transfer.

2. Label- a label is a symbolic name given to a specific memory location or instruction in the program. Labels are used to mark locations in the code that you may want to reference or jump to later in the program. They are essential for creating control flow structures like loops and conditional branches and for defining data and variables.

3. Operand: In assembly language programming, an operand is a value or data item on which an operation is performed. Operands can be constants, variables, or registers, and they are used in conjunction with assembly language instructions to perform various operations such as arithmetic calculations, data manipulation, and control flow.

4. Mnemonic: In assembly, each mnemonic represents a machine instruction. An example of one of these machine instructions add. Mul, and CMP are some other examples.

5. Registers: The processor can operate on numeric values (numbers) but must first save somewhere. The data is now kept in memory i.e. registers.

6. Macro: In assembly language, a macro is a named sequence of assembly instructions that can be defined once and then used multiple times throughout a program. Macros are similar to functions or procedures in high-level programming languages, but they are expanded at compile-time or assembly-time rather than executed at runtime. Macros are often used to simplify code, make it more readable, and reduce redundancy.

Here's a simple example of a macro in assembly language using the x86 architecture and NASM (Netwide Assembler) syntax:

```
%macro ADD_TWO_NUMBERS 2   ;      Define a macro named ADD_TWO_NUMBERS
with two arguments

   mov eax, %1   ;                Move the first argument into the EAX register

   add eax, %2    ;               Add the second argument to EAX

%endmacro
```

Here, we've defined a macro named ADD_TWO_NUMBERS that takes two arguments, %1 and %2. The %1 and %2 placeholders are used to represent the arguments passed to the macro. Inside the macro, we have two assembly instructions:

- mov eax, %1: This instruction moves the value of the first argument (%1) into the EAX register.
- add eax, %2: This instruction adds the value of the second argument (%2) to the value already in the EAX register.

7. **Directive:** Directives are used to define data, specify memory locations, and control the assembly process. Here are some common assembly directives explained in a simple way:

- .data: This directive is used to declare data sections where you define variables and their initial values. For example, you might use it to define integers, strings, or arrays.

```
.data

myVar   DWORD   42  ;    Declare a integer variable with value 42

myStr   DB     "Hello, World!"  ;     Declare a string variable
```

- .text: This directive is used to declare the code section where you write your actual assembly instructions. It tells the assembler that the following lines contain executable code.

```
   .text
MOV EAX, 42   ;      Move the value 42 into the EAX register
ADD EAX, 8     ;            Add 8 to the value in EAX
```

- .equ: This directive is used to define constants or symbolic names for values, making the code more readable and maintainable.

```
.equ MAX_VALUE, 100  ; Define a constant MAX_VALUE with a value of 100
```

Design of the Assembler & Assembler Design Criteria

The design of an assembler is a crucial aspect of developing a computer programming tool that translates assembly language code into machine code or binary code that can be executed by a computer's central processing unit (CPU). An assembler plays a fundamental role in the software development process, as it bridges the gap between human-readable assembly language and the computer's binary language. To create an effective assembler, designers must consider various criteria to ensure it performs its tasks accurately and efficiently. Let's elaborate on both the concept of assembler design and the criteria involved:

1. Assembler Design:

An assembler is a software program responsible for converting assembly language instructions (mnemonics) into machine code, which consists of binary values that can be directly executed by a computer's CPU. The design of an assembler typically involves the following components:

- Lexical Analysis: This is the initial phase where the assembler breaks down the assembly code into tokens, such as mnemonics, labels, operands, and comments. It identifies the structure of the code and organizes it for further processing.

- Parsing: In this stage, the assembler parses the tokens to understand the syntax and semantics of the assembly language. It checks for errors and ensures that the code follows the rules of the assembly language.

- Symbol Table Management: Assemblers need to manage symbols, including labels and variables used in the code. They create and maintain a symbol table that associates these symbols with memory addresses or values.

- Code Generation: This is the core of the assembler's functionality, where it translates assembly language instructions into machine code. It must consider the instruction set architecture (ISA) of the target computer to generate correct and efficient binary code.

- Error Handling: Effective error detection and reporting are essential in assembler design. When the assembler encounters syntax errors, undefined symbols, or other issues, it should provide clear and informative error messages to aid programmers in debugging.

- Output Generation: The assembler generates an output file containing the machine code or binary representation of the program. This file is typically in a format that can be loaded and executed by the computer.

2. Assembler Design Criteria:

When designing an assembler, developers must consider several criteria to ensure its functionality and usability. These criteria include:

- Correctness: The assembler must accurately translate assembly language code into machine code without introducing errors or altering the program's behavior.

- Efficiency: The assembler should generate efficient machine code that minimizes execution time and memory usage. It should also be optimized for speed and resource utilization during the assembly process.

- Portability: An assembler should be designed to support different computer architectures or ISA to accommodate various target platforms.

- Extensibility: The ability to add custom instructions or support for new hardware features is crucial, as computer architectures evolve over time.

- User-Friendly: The assembler's user interface should be intuitive and provide features like syntax highlighting, code completion, and error checking to assist programmers in writing clean and error-free assembly code.

- Documentation: Comprehensive documentation, including user manuals and reference guides, should be provided to assist users in understanding and using the assembler effectively.

- Performance Analysis: Tools for profiling and analyzing the generated code's performance can be beneficial for programmers seeking to optimize their programs.

Types of Assemblers-

Before going to discuss the types of assemblers, let’s understand forward referencing in assembly language-
Forward referencing in the context of assemblers refers to the situation where a symbol (like a label or variable) is referenced before it is defined in the program. Assemblers typically read code linearly from top to bottom. When encountering forward references, they might face difficulty in resolving the address or value associated with that symbol.
For instance, consider the following assembly code:
Example-

```
JMP Label1
...
...
...
Label1: NOP
```

Here, the assembler encounters the instruction JMP Label1 before it encounters the actual definition of Label1: NOP. The assembler needs to know the memory address of Label1 to correctly generate

the machine code for the JMP instruction. Forward referencing becomes an issue because the assembler doesn't know the memory address of Label1 until it has processed the entire program. To handle forward referencing, multi-pass assemblers utilize multiple passes through the source code. In the first pass, they collect all the symbols and their corresponding memory addresses or values, and in the subsequent passes, they use this information to resolve any forward references, thereby producing the correct machine code. This approach enables the assembler to handle more complex code structures and dependencies.

Types of Assemblers-

There are generally two types of assemblers:

One-pass assemblers: One-pass assemblers read the source code only once, converting it directly into machine code. They are efficient but have limitations, such as the inability to resolve forward references (when a variable or label is used before it is defined).

Two-pass assemblers: Two-pass assemblers read the source code twice. During the first pass, they collect all the information about the code, such as labels and variables, and during the second pass, they use this information to generate the machine code. They can handle forward references and are more versatile but are generally slower than one-pass assemblers.

First Pass of a Two Pass Assembler: A first pass assembler is a program that reads the source code of a computer program written in assembly language and it collects information about the symbols in code, such as labels, variables or constants and it also calculates memory addresses for these elements. First pass prepares a symbol table, which keeps track of the memory addresses assigned to each label or variable in the code.

Design of working of Assembler

1. Analysis Phase (PASS 1 of Assembler):

- To build symbol table for synthesis phase to proceed.
- Determines address of each symbols called as memory allocation.
- Location counter used to hold address of next instruction.
- Isolated label, mnemonic opcode, operands, constants etc.
- Validate meaning & address of each statements.

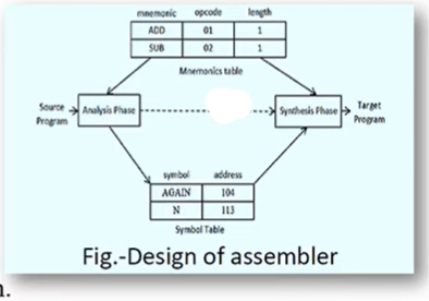
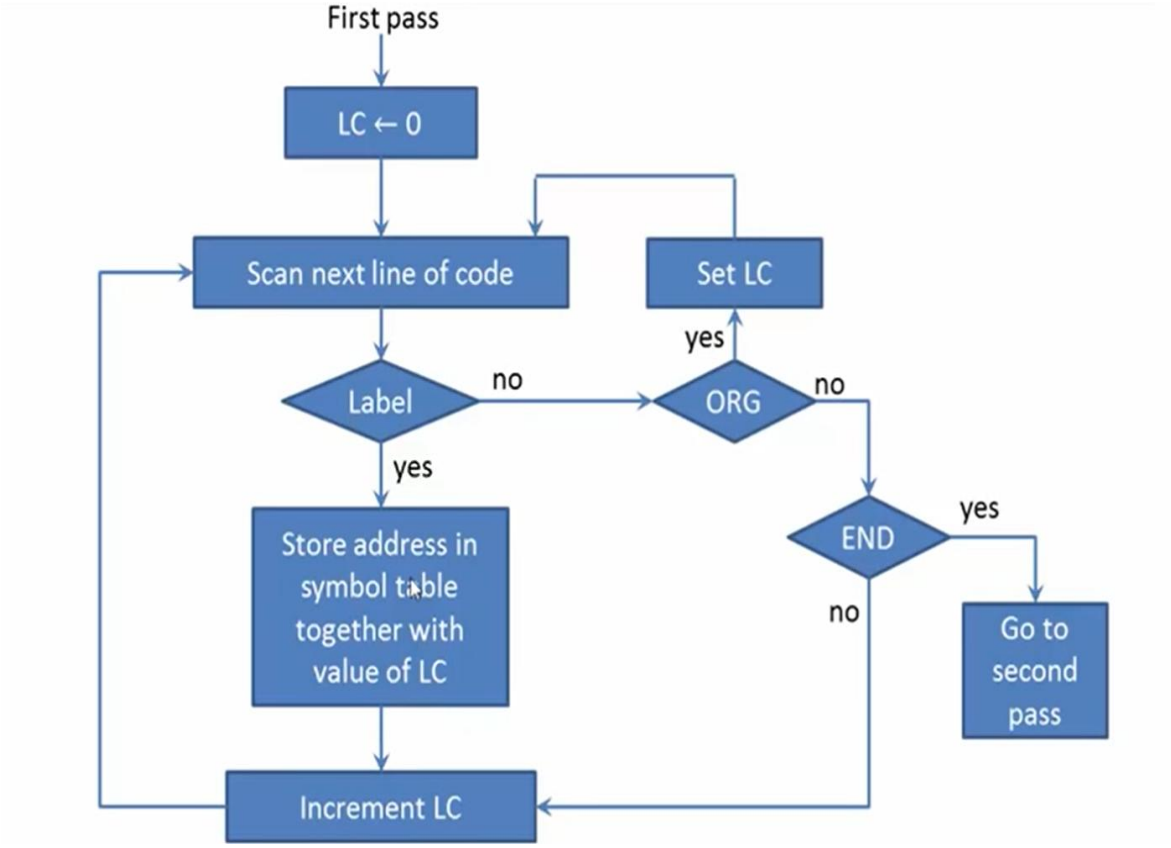


Fig.-Design of assembler

2. Synthesis Phase: (PASS 2 of Assembler):

- Use data structures generated by analysis phase.
- To build machine instructions for every assembly statements as per mnemonic code & there address allocation.
- Synthesis machine instruction as per source code.



PASS 1 of Assembler (Analysis Phase)

Data Structures in Assembly Language

1. **Symbol Table (ST or SYMTAB):** Store value or address assign to the Label.

Label	Address
JOHN	200
L1	202
X	204

2. **Literal Table (LT or LITAB):** Store each literals or constants (= '3')with its location.

Index	Literal	Address
0	='3'	200
1	='2'	202

```
[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
```

PASS 1 of Assembler (Analysis Phase)

Data Structures in Assembly Language

3. Operation Code Table(OPTAB): Store Mnemonic operation code with there opcodes & length.

Inst.	Opcode	Length(Bytes)
MOVER	3	2
MOVEM	X	1
MOVER	2	2

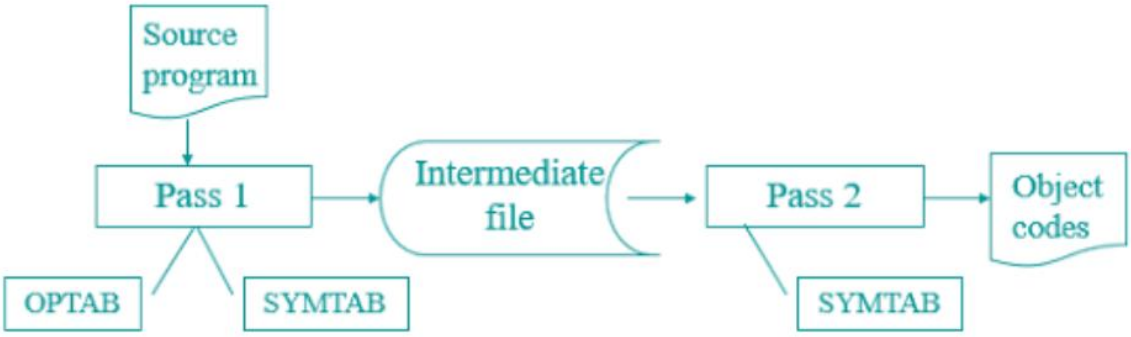
```
[Label] [Opcode] [operand]

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
```

Second Pass of a Two Pass Assembler:

During the second pass, it generates the actual machine code. This approach allows the assembler to resolve any forward references (such as labels that are used before they are defined) in the source code.

PASS 2 of Assembler (Synthesis Phase)

- In the second pass the instructions are again read and are assembled using the symbol table.
- Basically, the assembler goes through the block of program 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.
- Convert mnemonic operations code, symbolic table operands with there equivalent machine code.
- Convert data constants to internal machine representations.
- Convert complete program into .obj file.



Advantages & Disadvantages of Assembly Language Program

Advantages:

1. Hardware oriented.
2. Increasing readability by using data structure.
3. Useful in embedded system.
4. Less resources, managing size & code.
5. Access hardware drivers & system code easily as compare to high level lang.

Disadvantages:

1. Machine dependent.
2. Platform dependent, Porting to another platform is not easy.
3. More difficult to debug.
4. More complex in nature.

Macro and Macro Processors

In assembly language, a macro is a sequence of instructions or directives that is given a name. This allows you to use that sequence of instructions by referring to its name, making it easier to write and manage complex code. A macro processor is a tool that expands these macro definitions into actual assembly code before the code is assembled or compiled. This helps in simplifying the coding process and making it more efficient.

OR

To process macro instructions, most assembler use preprocessors known as macro processors.

Let's understand this with an example:

Suppose you have a repetitive task in your assembly code that involves loading a value into a register, performing some operation, and then storing the result. Instead of writing these instructions multiple times, you can define a macro for this sequence.

Here is an example of how you can define and call a simple macro in assembly language:

Macro Definition:

```
;     Macro definition
MACRO add_sequence num1, num2, num3
   MOV AX, num1  ; Move num1 into AX register
   ADD AX, num2  ; Add num2 to AX
   MOV num3, AX  ; Move the result to the memory location num3
ENDM
```

Here, the add_sequence macro takes three arguments (num1, num2, and num3) and performs the sequence of operations on them. You can then call this macro with specific values for num1, num2, and num3 in your code. The macro processor will replace the macro call with the actual sequence of instructions during the assembly process.

- add_sequence is the name of the macro that you defined earlier.
- num1, num2, and num3 are the arguments that you pass to the macro. These are values or variables that will be used within the macro sequence.

Macro Call-

```
;   Define the macro for calculating the square of a number
;    Input: R1 contains the number
;   Output: R2 contains the square of the number

MACRO   MACRO_CALCULATE_SQUARE

MOV R2, R1      ; Move the number to be squared to R2
 MUL R2, R1      ; Multiply R2 by itself to get the square
ENDM

;   Assume that R1 contains the number we want to find the square of
MOV R1, 5    ; Assign 5 to the R1 register

; Call the macro to calculate the square of the number in R1
MACRO_CALCULATE_SQUARE

; The result will be stored in R2 after the macro call
```

Macro Expansion-

In assembly language, macro expansion is a mechanism that allows you to define a set of instructions or a sequence of operations as a macro and then use that macro throughout your program as if it were a single instruction. This can help in simplifying the code and making it more readable and maintainable.

Here's a simple example to illustrate how macro expansion works:

Suppose you want to repeatedly add two numbers and store the result in a specific register. You can create a macro for this operation as follows:

```
;    Macro definition
MACRO  add_numbers num1, num2, result
   MOV result, num1
   ADD result, num2
ENDM

;    Using the macro. Here main is not predefined or default keyword but a label
main:
   add_numbers 5, 3, R0  ; This line will expand to MOV R0, 5 followed by ADD R0, 3
   ;    other instructions
```

In this example, the add_numbers macro takes three parameters: num1, num2, and result. It then expands to the appropriate assembly instructions (MOV and ADD) when it's used in the main section of the program. This way, you can avoid writing the same set of instructions multiple times, making your code more concise and easier to understand.

During the assembly process, the assembler replaces the macro call with the corresponding set of instructions defined in the macro, effectively expanding it inline. This makes it easier to manage and modify complex or repetitive code by encapsulating it within a single macro definition.

Difference between Macro and Subroutine or Function or Procedure

Count	Macro	Subroutine
1	Macro name in the mnemonic field leads to expansion only	It is a call statement in the program That leads to execution.
2	Macros are handled by Assembler (preprocessor) during assembly time.	Completely handled by hardware at run time
3	Macro definition and expansion are done by assembler. Therefore, assembler must know all the features, exceptions and options associated with them.	
4	The hardware knows nothing about macros	Assembler knows nothing about subroutines
5	Macro processing increase the size of resulting code but results in	Smaller in size but has substantial

	faster execution of program	overhead of control transfer during execution
6	Debugging in macro is difficult as compared to subroutine as size of the code grows	Debugging is comparatively easy.
7	Macro are used for small task	Subroutines are used for complex task
8	Does not alter the flow of execution	It alters the flow of execution
9	Macro call a process at translation time or assembly time	Subroutine calls a process at run time
10	Preprocessed by the preprocessor	Compiled separately