CS202 – System Software

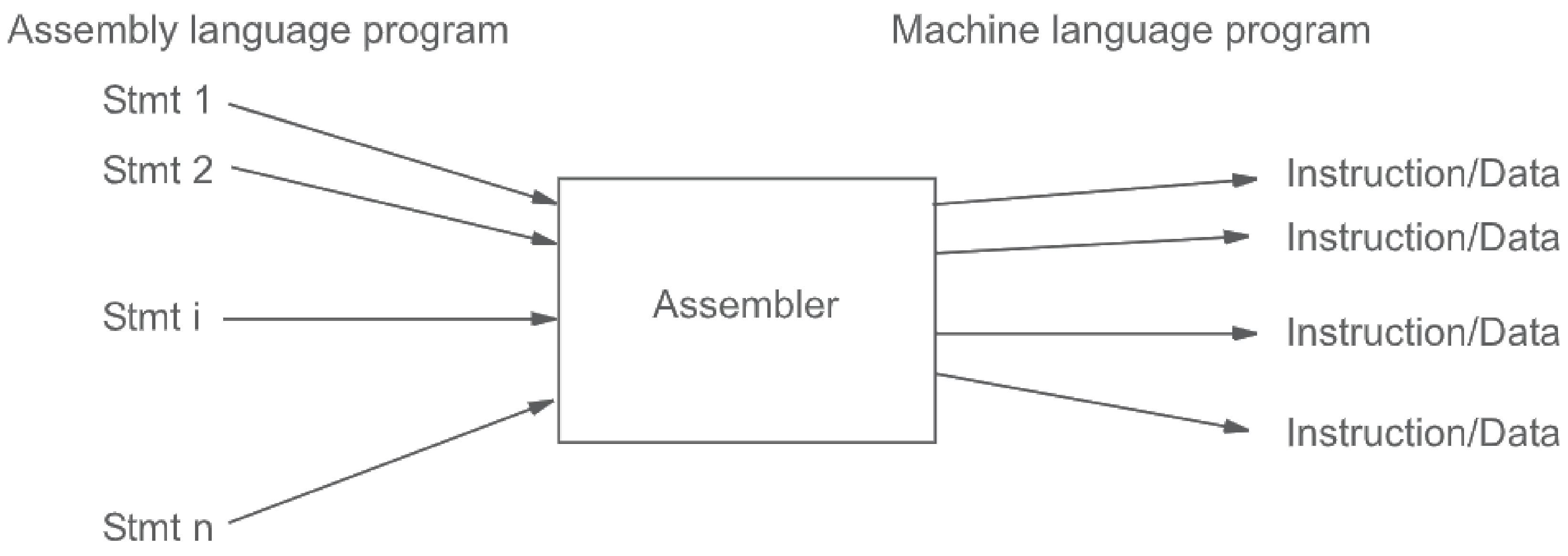
Dr. Manish Khare

Assembler

- Assembler is the tool to convert an assembly language program into machine language one, understandable by the processor that executes it.

- The complexity of the process depends upon various factors, including size of the instruction set, different addressing modes supported, length of program being translated, etc.

- Ideally, an assembler can be viewed as a tool that looks at each statement of the assembly language program and generates the corresponding machine language counterpart.
- This has been depicted in following Fig.



- However, apart from producing code in the machine language, some more information is needed to facilitate loading of program at arbitrary start address in the memory.
- It needs corrections to the address sensitive values, like operand addresses, jump targets, etc. The assemblers normally generate code starting at offset zero.
- For multi-section programs (that is, programs with multiple code and/or data sections), each section is assembled starting at offset zero, so that maximum flexibility exists regarding their loading into the memory for execution.
- Thus, information about all the sections and their attributes is also needed to produce the final executable version of the program.

Simple Manual Assembler

- Consider a simple hypothetical accumulator based processor. All memory load and store operations are through the accumulator register (call it 4).

- Arithmetic and logic operations mostly use A as a source register and also the destination register. Assume that there are two more 32-bit general purpose registers B, C, and an index register I.

- Arithmetic is permitted on I also. The addressing modes supported ate the immediate and indirect through the index register.

- All memory addresses are 32-bit wide. Register-to-register data movement is also supported. The instructions and the associated codes are as shown in Table.

- This table depicting the machine instructions, is called the Machine Opcode Table (MOT).

Instruction	Size (in bytes)	Format	Meaning
MVI A, <*constant*>	5	<0, 4 byte constant>	$A \leftarrow$ *constant*
MVI B, <*constant*>	5	<1, 4 byte constant>	$B \leftarrow$ *constant*
MVI C, <*constant*>	5	<2, 4 byte constant>	$C \leftarrow$ *constant*
MVI I, <*constant*>	5	<3, 4 byte constant>	$I \leftarrow$ *constant*
LOAD <*constant*>	5	<4, 4 byte constant>	$A \leftarrow$ memory[<*constant*>]
STORE <*constant*>	5	<5, 4 byte constant>	memory[<*constant*>] $\leftarrow A$
LOADI	1	< 6 >	$A \leftarrow$ memory[I]
STORI	1	< 7 >	memory[I] $\leftarrow A$
ADD B	1	< 8 >	$A \leftarrow A + B$
ADD C	1	< 9 >	$A \leftarrow A + C$
MOV A, B	1	< 10 >	$A \leftarrow B$
MOV A, C	1	< 11 >	$A \leftarrow C$
MOV B, C	1	< 12 >	$B \leftarrow C$
MOV B, A	1	< 13 >	$B \leftarrow A$
MOV C, A	1	< 14 >	$C \leftarrow A$
MOV C, B	1	< 15 >	$C \leftarrow B$
INC A	1	< 16 >	$A \leftarrow A + 1$
INC B	1	< 17 >	$B \leftarrow B + 1$
INC C	1	< 18 >	$C \leftarrow C + 1$
CMP A, <*constant*>	5	<19, 4 byte constant>	compare A to *constant*
CMP B, <*constant*>	5	<20, 4 byte constant>	compare B to *constant*
CMP C, <*constant*>	5	<21, 4 byte constant>	compare C to *constant*
ADDI <*constant*>	5	<22, 4 byte constant>	$I \leftarrow I +$ *constant*
JE <*label*>	5	<23, 4 byte address>	jump on equal to <*label*>
JMP <*label*>	5	<24, 4 byte address>	jump to <*label*>
STOP	1	< 25 >	stop the processor

Designing Assembler

- An assembler is a program that receives as input an assembly language program and generates its machine language correspondent along with information for the linker/loader and report for the consumer.

- Fundamentally it converts code in one language into another. Assemblers are nothing more than huge translators that convert code from one language to another following recognized syntax rules.
- In many cases the code is converted into machine code with information concerning the program, position of internal variables that might be referenced by another program, information concerning local addresses that will require to be updated depending on where the operating system allocates the program to memory.

- These activities are the role of the linker/loader.

Designing Assembler

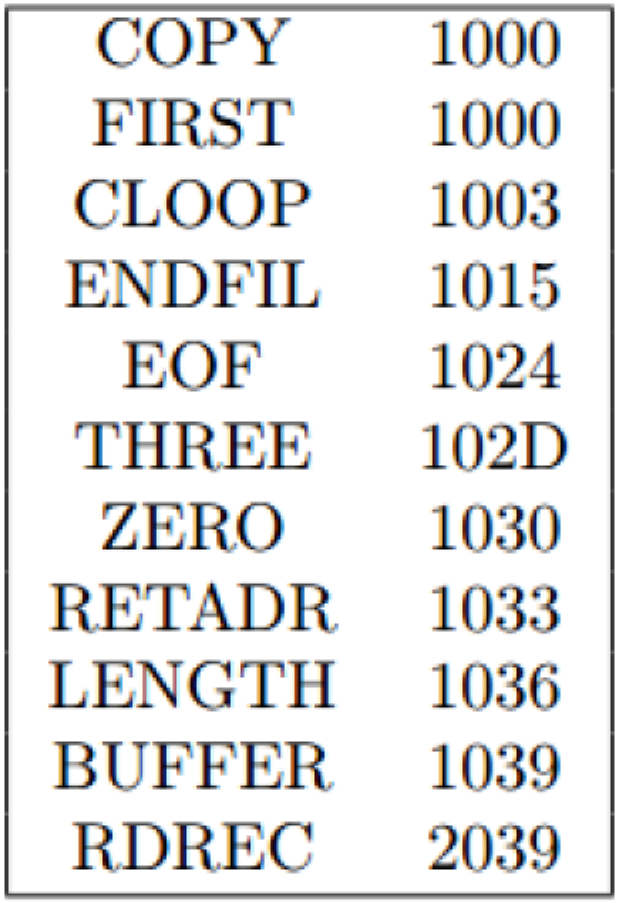
- **Addressing Options**
- ***Absolute Addresses:*** These are the address fields that will not require to be customized by the loader.
- ***Relocatable Addresses:*** These are the addresses that will require to be accustomed if and only if the loader choose to position this program at a dissimilar memory location than the assembler allocated.
- ***Externally defined Symbols/Labels:*** The assembler does not recognize the addresses (or values) of these symbols and it depends on the loader to locate the programs comprising these symbols, load them into memory, and position the addresses of these symbols in the calling program.
- ***Indirect Addresses:*** The address refers to a location in memory that comprises the target address.
- ***Direct Addresses:*** Address of an explicit position in the physical memory.
- ***Immediate:*** The normal address will be utilized for an immediate value instead of an address.

Data Structures used in Assembler

- The data structures used in the design of 2 pass algorithm are:
  - Operation Code Table (OPTAB)
  - Symbol Table (SYMTAB)
  - Location Counter(LOCCTR)

- **Operation Code Table (OPTAB)**
- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.
- In pass 1 the OPTAB is used to look up and validate the operation code in the source program and to find the instruction length for incrementing LOCCTR. In pass 2, it is used to translate the operation codes to machine language.

- **Symbol Table (SYMTAB)**
- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1,labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1. During Pass 2, symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions. SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. A sample SYMTAB is shown below.



COPY	1000
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	1024
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039

- **Location Counter (LOCCTR)**
- Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

Design Assembler

- Assembler generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code.
- Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler.
- Here assembler divide these tasks in two passes:

- **Pass-1:**
  - Define symbols and literals and remember them in symbol table and literal table respectively.
  - Keep track of location counter
  - Process pseudo-operations
- **Pass-2:**
  - Generate object code by converting symbolic op-code into respective numeric op-code
  - Generate data for literals and look for values of symbols

- Firstly, We will take a small assembly language program to understand the working in their respective passes. Assembly language statement format:

[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 take a look on how this program is working:

1. **START:** This instruction starts the execution of program from location 200 and label with START provides name for the program.(JOHN is name for program)
2. **MOVER:** It moves the content of literal(=’3’) into register operand R1.
3. **MOVEM:** It moves the content of register into memory operand(X).
4. **MOVER:** It again moves the content of literal(=’2’) into register operand R2 and its label is specified as L1.
5. **LTORG:** It assigns address to literals(current LC value).
6. **DS(Data Space):** It assigns a data space of 1 to Symbol X.
7. **END:** It finishes the program execution.

- **Working of Pass-1:** Define Symbol and literal table with their addresses.
  Note: Literal address is specified by LTORG or END.
- **Step-1: START 200** (here no symbol or literal is found so both table would be empty)
- **Step-2: MOVER R1, =’3′ 200** (=’3′ is a literal so literal table is made)

Literal	**Address**
=’3′	– – –

Step-3: MOVEM R1, X 201

X is a symbol referred prior to its declaration so it is stored in symbol table with blank address field.

Symbol	Address
X	–––

➢ Step-4: L1 MOVER R2, =’2’ 202

L1 is a label and =’2’ is a literal so store them in respective tables

Symbol	Address
X	– – –
L1	202

Literal	Address
=’3′	– – –
=’2′	– – –

Step-5: LTORG 203

Assign address to first literal specified by LC value, i.e., 203

Literal	Address
=’3′	203
=’2′	– – –

Step-6: X DS 1 204

It is a data declaration statement i.e X is assigned data space of 1. But X is a symbol which was referred earlier in step 3 and defined in step 6.This condition is called Forward Reference Problem where variable is referred prior to its declaration and can be solved by back-patching. So now assembler will assign X the address specified by LC value of current step.

Symbol	Address
X	204
L1	202

➢ Step-7: END 205

Program finishes execution and remaining literal will get address specified by LC value of END instruction. Here is the complete symbol and literal table made by pass 1 of assembler.

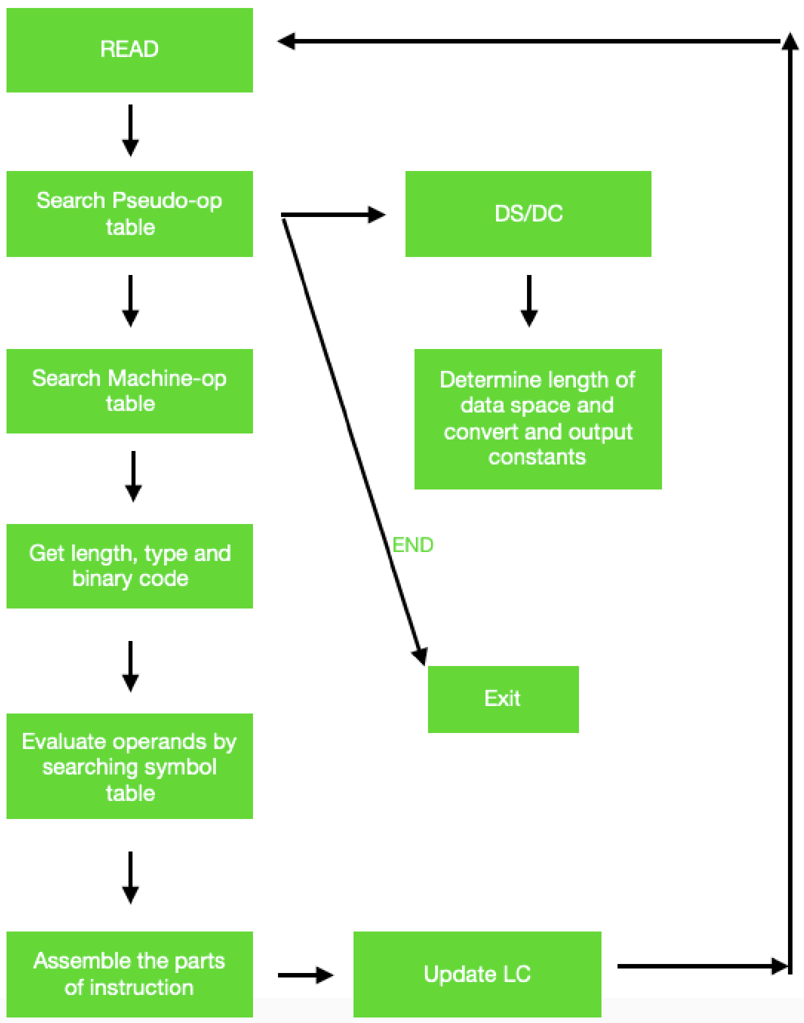
Symbol	Address
X	204
L1	202

Literal	Address
=’3′	203
=’2′	205

- Now tables generated by pass 1 along with their LC value will go to pass-2 of assembler for further processing of pseudo-opcodes and machine op-codes.

- **Working of Pass-2:**
  Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration(machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(pseudo-op table).
- Various Data bases required by pass-2:

```
1.MOT table(machine opcode table)
2.POT table(pseudo opcode table)
3.Base table(storing value of base register)
4.LC ( location counter)
```


READ
Search Pseudo-op table
DS/DC
Search Machine-op table
Determine length of data space and convert and output constants
Get length, type and binary code
END
Exit
Evaluate operands by searching symbol table
Assemble the parts of instruction
Update LC

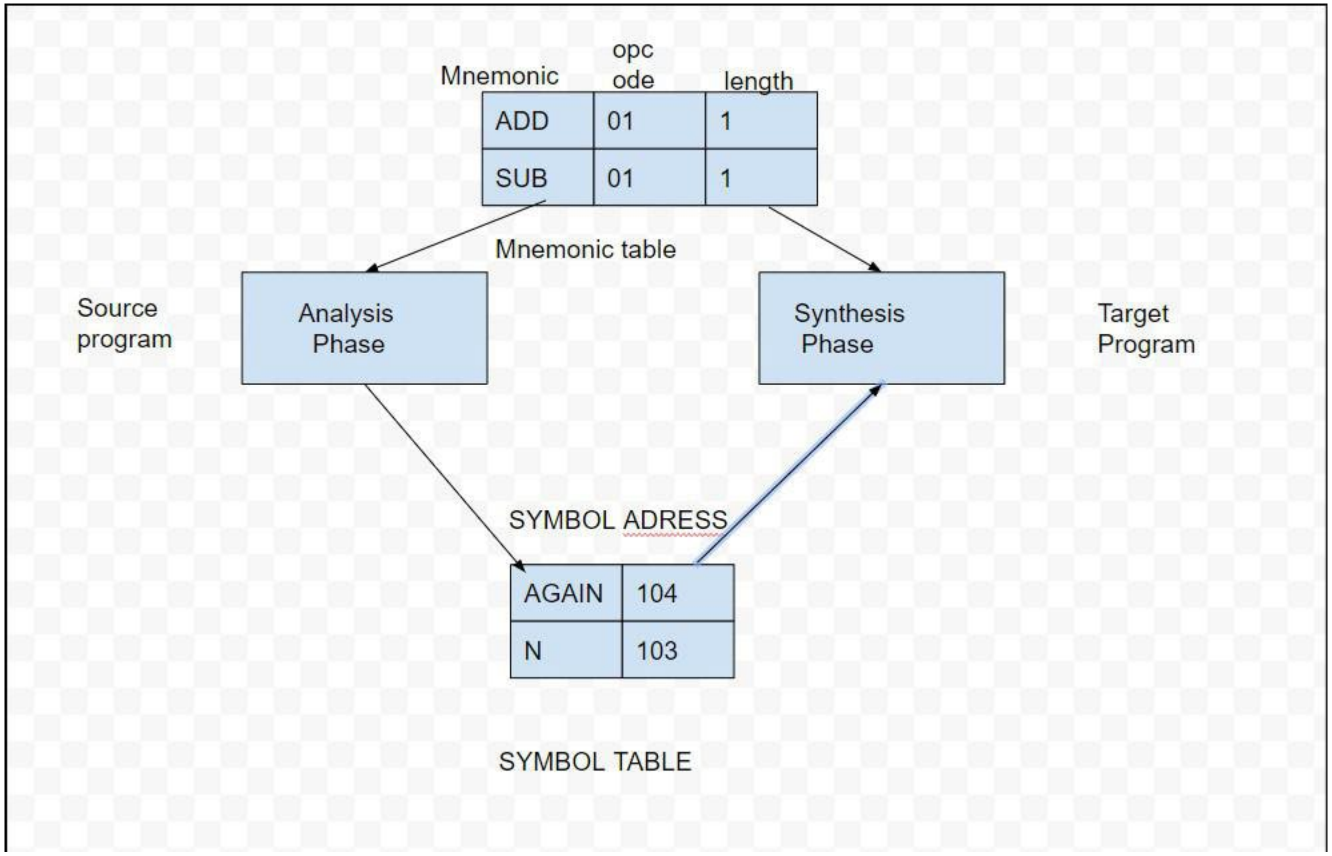
Phases of Assembler

- The Assembler is a program that converts assembly language into machine language that can be executed by a computer.
- The Assembler operates in two main phases: Analysis Phase and Synthesis Phase.
  - The Analysis Phase validates the syntax of the code, checks for errors, and creates a symbol table.
  - The Synthesis Phase converts the assembly language instructions into machine code, using the information from the Analysis Phase.
- These two phases work together to produce the final machine code that can be executed by the computer.
- The combination of these two phases makes the Assembler an essential tool for transforming assembly language into machine code, ensuring high-quality and error-free software.

- **1) Analysis Phase**

1. The primary function performed by the analysis phase is the building of the symbol table. It must determine the memory address with which each symbolic name used in a program is associated in the assembly program of the address of N would be known only after fixing the addresses of all program elements-whether instructions or memory areas-that preceding it. This function is called memory allocation.
2. Memory allocation is performed by using a data structure called location counter (LC).
3. The analysis phase ensures that the location counter always contains the address that the next memory word in the target program should have
4. At the start of its processing, it initializes the location counter to the constant specified in the START statement.

5. While processing a statement, it checks whether the statement has a label.
6. If so, it enters the label and the address contained in the location counter in a new entry of the symbol table. It then finds how many memory words are needed for the instruction or data represented by the assembly statement and updates the address in the location counter by that number. (Hence the word ‘counter’ in location counter”.).
7. The amount of memory needed for each assembly statement depends on the mnemonic of an assembly statement. It obtains this information from the length field in the mnemonics table.
8. It obtains this information from the length field in the mnemonics table. For DC and DS statements, the memory requirement further depends on the constant appearing in the operand field, so the analysis phase should determine it appropriately.
9. We use the notation **<LC>** for the address contained in the location counter.
10. The Symbol table is constructed during analysis and used during synthesis.



Mnemonic	opc ode	length
ADD	01	1
SUB	01	1

SYMBOL	ADRESS
AGAIN	104
N	103