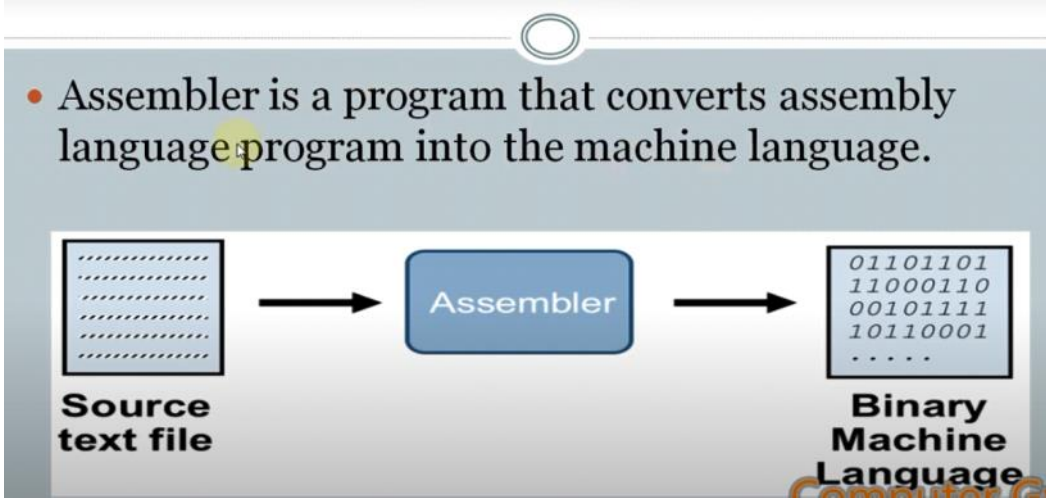
Elements of Assembly Language Programming

Tuesday, January 18, 2022 3:17 PM

SYSTEM SOFTWARE

- ELEMENTS OF ASSEMBLY LANGUAGE PROGRAMMING

Screen clipping taken: 1/19/2022 2:08 PM



- Assembler is a program that converts assembly language program into the machine language.

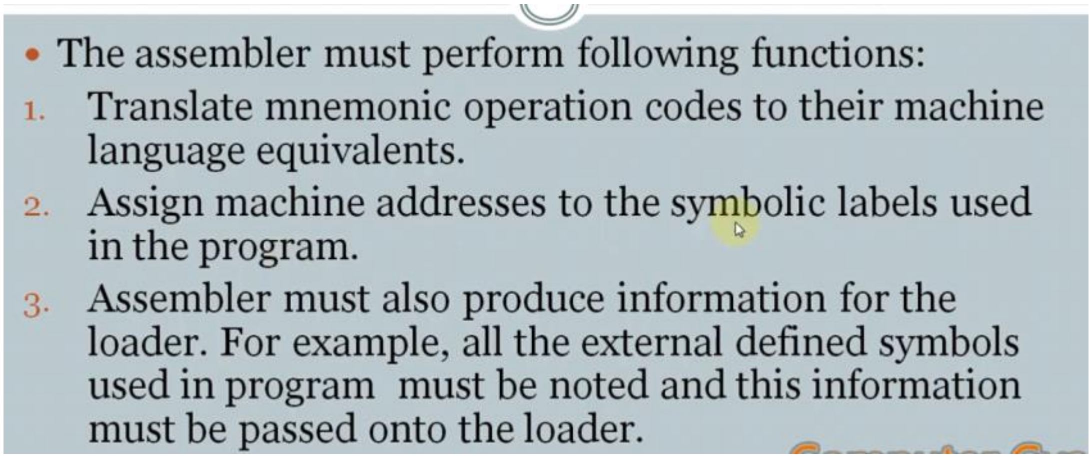
Source text file → Assembler → Binary Machine Language

```
01101101
11000110
00101111
10110001
. . . . .
```

Screen clipping taken: 1/19/2022 2:09 PM

✓Device drivers

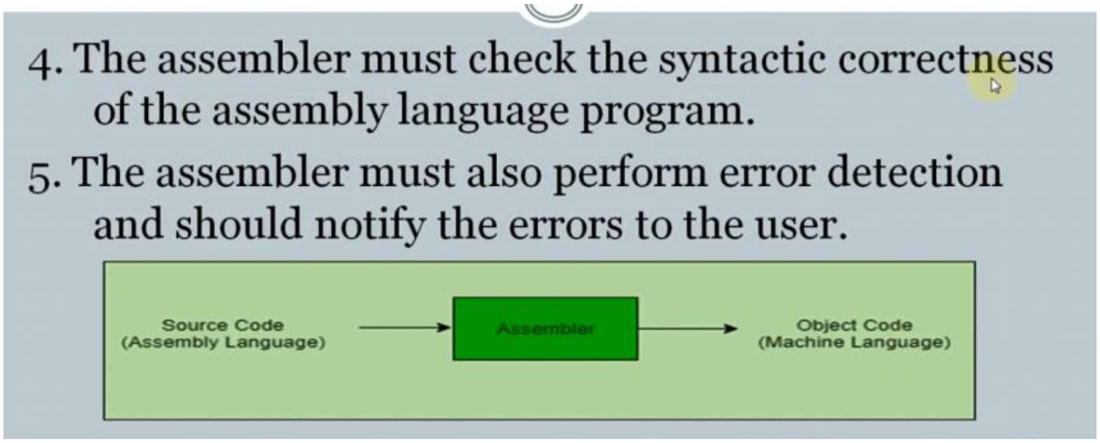
Screen clipping taken: 1/19/2022 2:58 PM



- The assembler must perform following functions:

1. Translate mnemonic operation codes to their machine language equivalents.
2. Assign machine addresses to the symbolic labels used in the program.
3. Assembler must also produce information for the loader. For example, all the external defined symbols used in program must be noted and this information must be passed onto the loader.

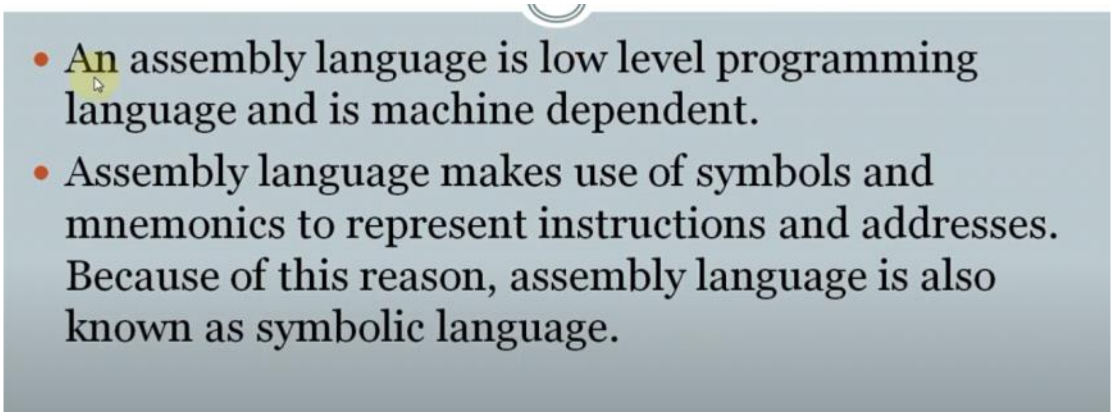
Screen clipping taken: 1/19/2022 2:11 PM



4. The assembler must check the syntactic correctness of the assembly language program.
5. The assembler must also perform error detection and should notify the errors to the user.

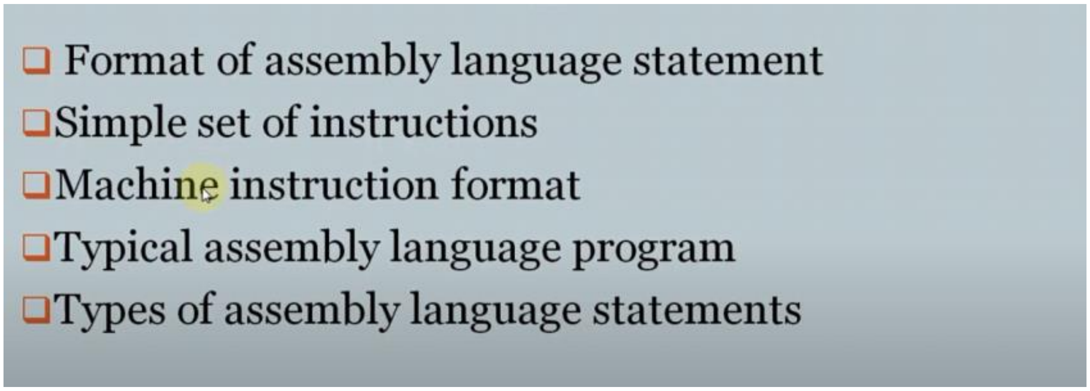
Source Code (Assembly Language) → Assembler → Object Code (Machine Language)

Screen clipping taken: 1/19/2022 2:11 PM



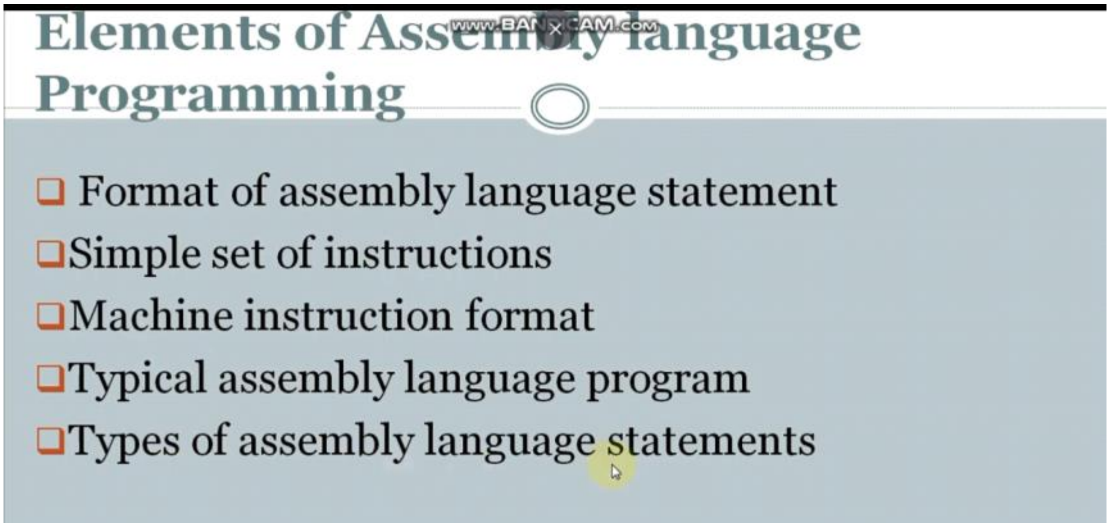
- An assembly language is low level programming language and is machine dependent.
- Assembly language makes use of symbols and mnemonics to represent instructions and addresses. Because of this reason, assembly language is also known as symbolic language.

Screen clipping taken: 1/19/2022 2:12 PM



- ❑ Format of assembly language statement
- ❑ Simple set of instructions
- ❑ Machine instruction format
- ❑ Typical assembly language program
- ❑ Types of assembly language statements

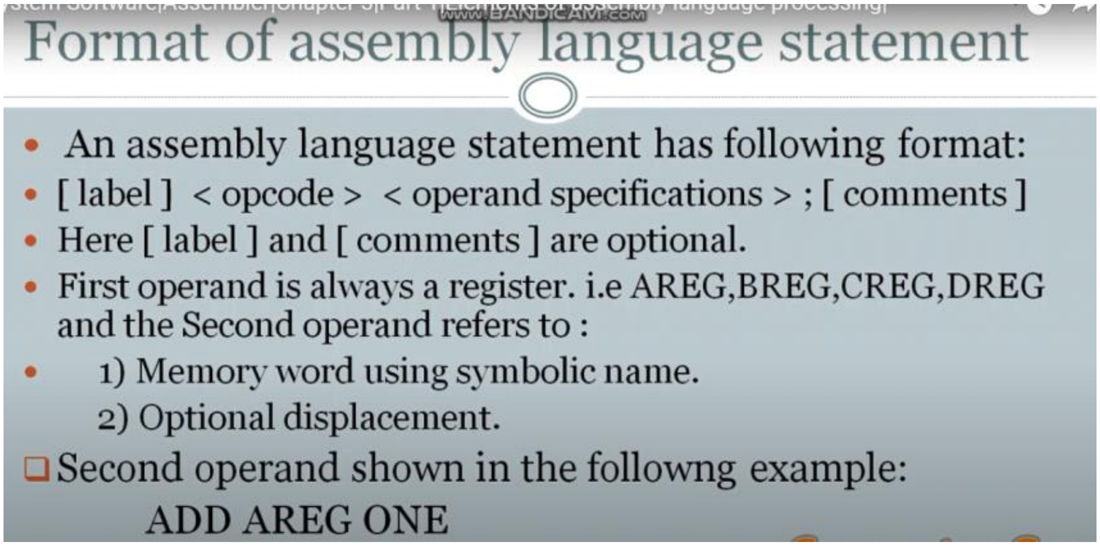
Screen clipping taken: 1/19/2022 2:13 PM



Elements of Assembly language Programming

- ❑ Format of assembly language statement
- ❑ Simple set of instructions
- ❑ Machine instruction format
- ❑ Typical assembly language program
- ❑ Types of assembly language statements

Screen clipping taken: 1/19/2022 2:13 PM



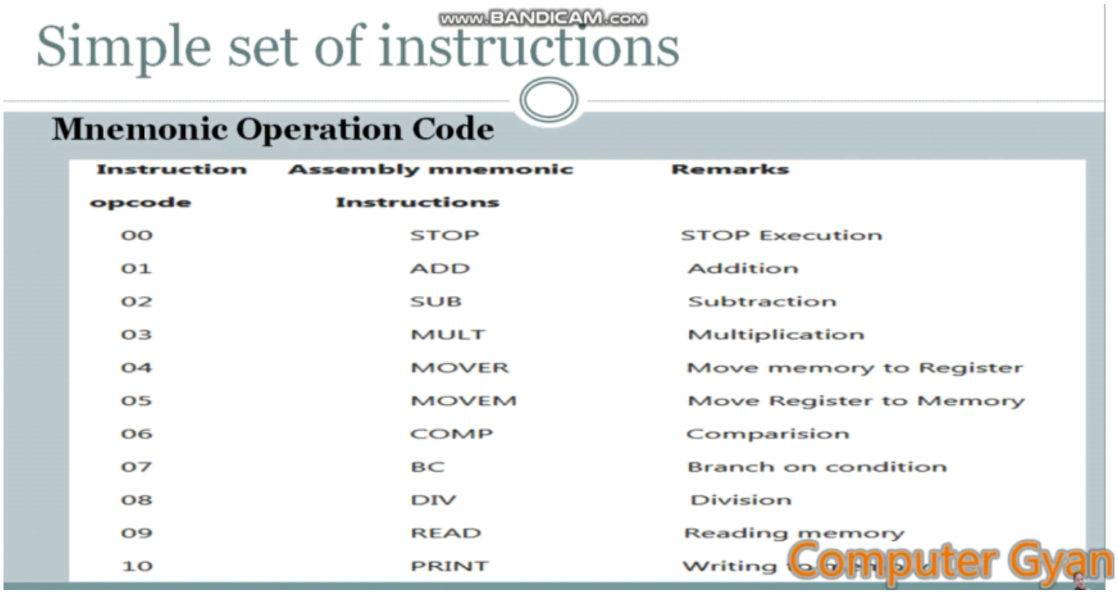
Format of assembly language statement

- An assembly language statement has following format:
- [label] < opcode > < operand specifications > ; [comments]
- Here [label] and [comments] are optional.
- First operand is always a register. i.e AREG,BREG,CREG,DREG and the Second operand refers to :
- 1) Memory word using symbolic name.
  2) Optional displacement.

☐ Second operand shown in the followng example:

ADD AREG ONE

Screen clipping taken: 1/19/2022 2:14 PM

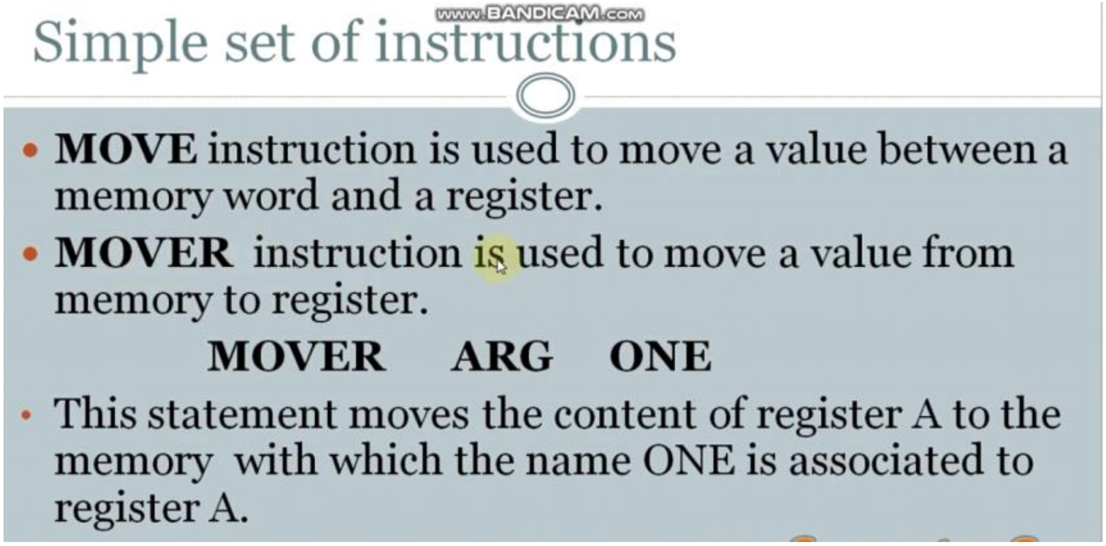


Simple set of instructions

Mnemonic Operation Code

Instruction opcode	Assembly mnemonic Instructions	Remarks
00	STOP	STOP Execution
01	ADD	Addition
02	SUB	Subtraction
03	MULT	Multiplication
04	MOVER	Move memory to Register
05	MOVEM	Move Register to Memory
06	COMP	Comparision
07	BC	Branch on condition
08	DIV	Division
09	READ	Reading memory
10	PRINT	Writing [illegible]

Screen clipping taken: 1/19/2022 2:17 PM



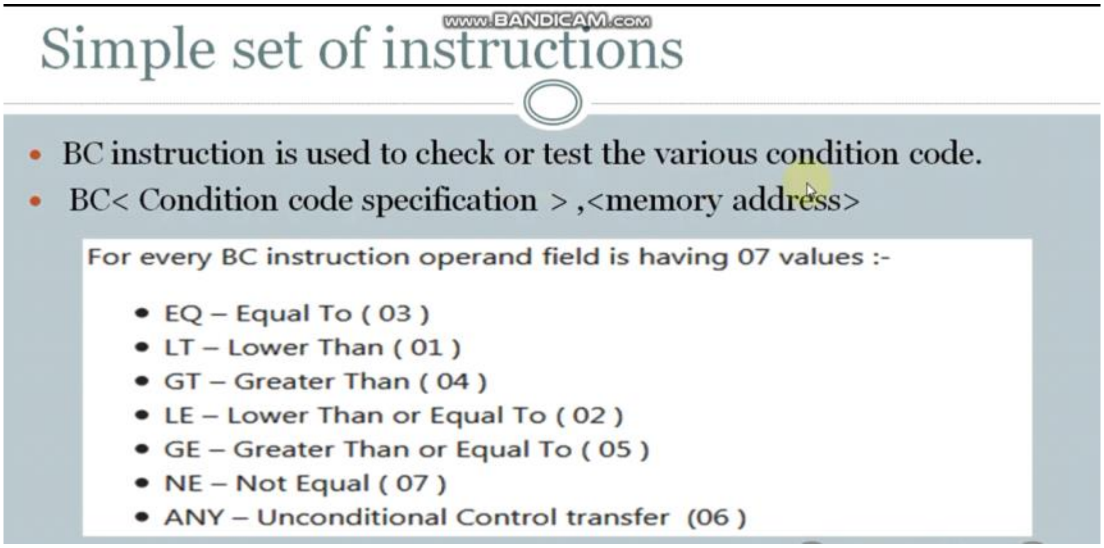
Simple set of instructions

- **MOVE** instruction is used to move a value between a memory word and a register.
- **MOVER** instruction is used to move a value from memory to register.

MOVER ARG ONE

- This statement moves the content of register A to the memory with which the name ONE is associated to register A.

Screen clipping taken: 1/19/2022 2:19 PM



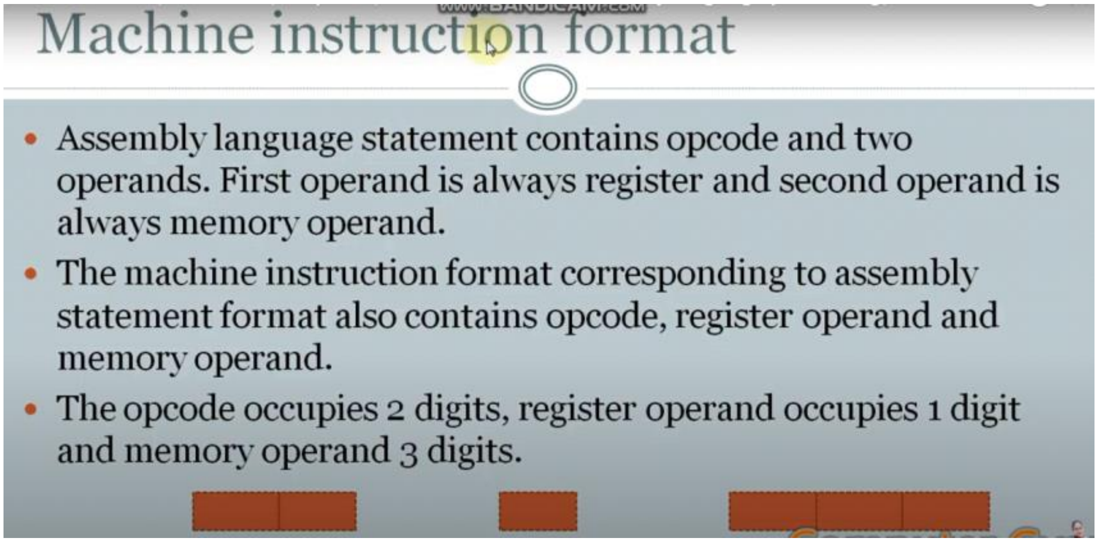
Simple set of instructions

- BC instruction is used to check or test the various condition code.
- BC< Condition code specification > ,<memory address>

For every BC instruction operand field is having 07 values :-

- EQ – Equal To (03)
- LT – Lower Than (01)
- GT – Greater Than (04)
- LE – Lower Than or Equal To (02)
- GE – Greater Than or Equal To (05)
- NE – Not Equal (07)
- ANY – Unconditional Control transfer (06)

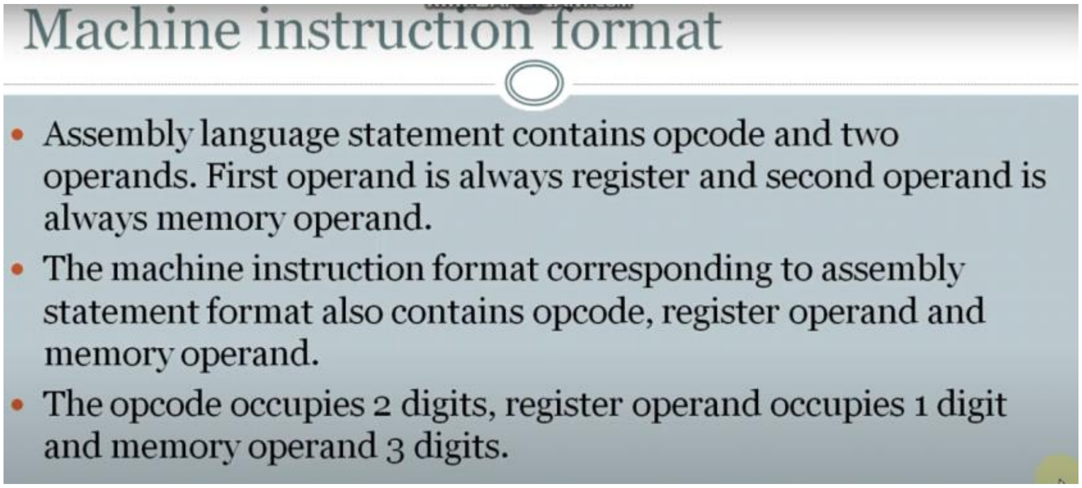
Screen clipping taken: 1/19/2022 2:20 PM



Machine instruction format

- Assembly language statement contains opcode and two operands. First operand is always register and second operand is always memory operand.
- The machine instruction format corresponding to assembly statement format also contains opcode, register operand and memory operand.
- The opcode occupies 2 digits, register operand occupies 1 digit and memory operand 3 digits.

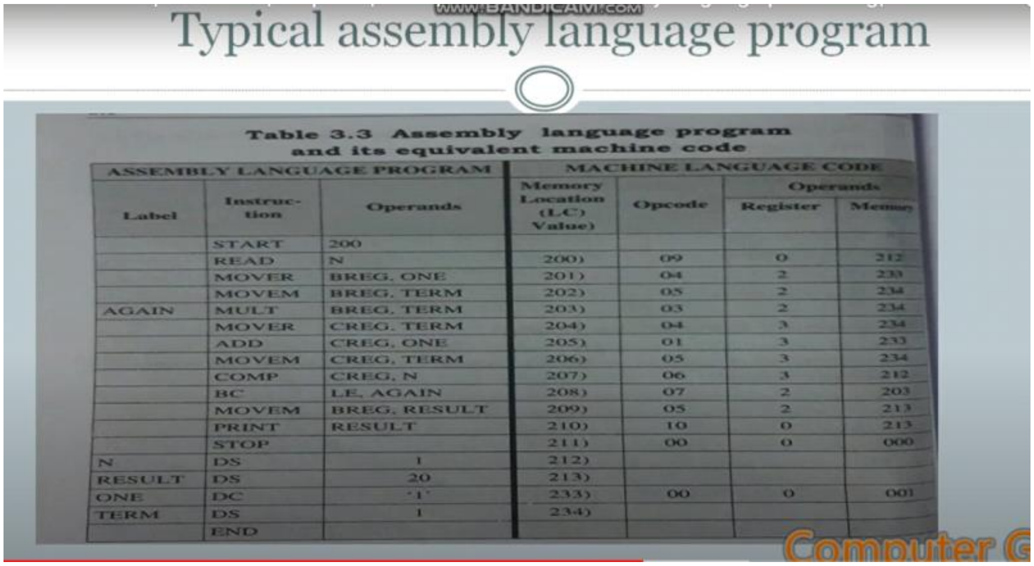
Screen clipping taken: 1/19/2022 2:21 PM



Machine instruction format

- Assembly language statement contains opcode and two operands. First operand is always register and second operand is always memory operand.
- The machine instruction format corresponding to assembly statement format also contains opcode, register operand and memory operand.
- The opcode occupies 2 digits, register operand occupies 1 digit and memory operand 3 digits.

Screen clipping taken: 1/19/2022 2:22 PM

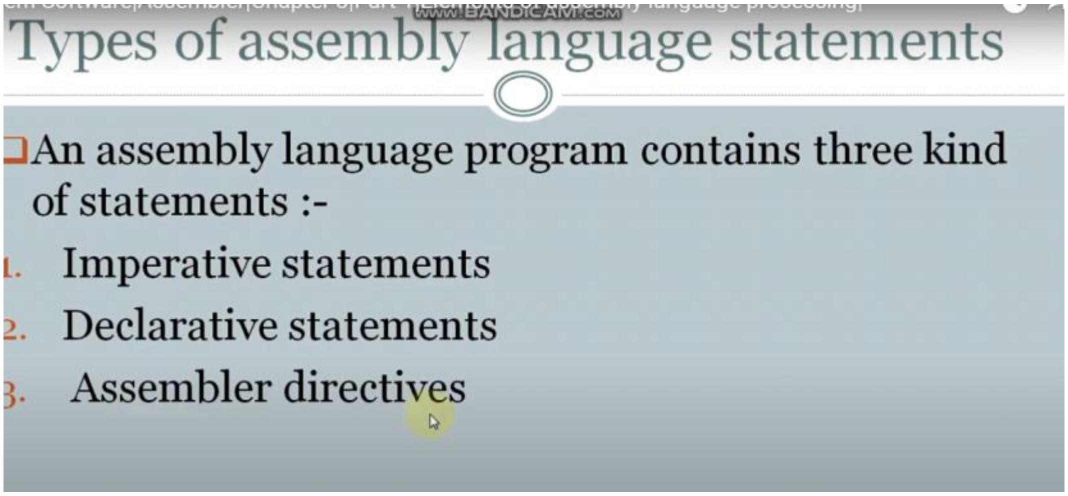


Typical assembly language program

Table 3.3 Assembly language program and its equivalent machine code

ASSEMBLY LANGUAGE PROGRAM			MACHINE LANGUAGE CODE			
Label	Instruction	Operands	Memory Location (LC Value)	Opcode	Operands	
					Register	Memory
	START	200				
	READ	N	200)	09	0	212
	MOVER	BREG, ONE	201)	04	2	233
	MOVEM	BREG, TERM	202)	05	2	234
AGAIN	MULT	BREG, TERM	203)	03	2	234
	MOVER	CREG, TERM	204)	04	3	234
	ADD	CREG, ONE	205)	01	3	233
	MOVEM	CREG, TERM	206)	05	3	234
	COMP	CREG, N	207)	06	3	212
	BC	LE, AGAIN	208)	07	2	203
	MOVEM	BREG, RESULT	209)	05	2	213
	PRINT	RESULT	210)	10	0	213
	STOP		211)	00	0	000
N	DS	1	212)			
RESULT	DS	20	213)			
ONE	DC	'1'	233)	00	0	001
TERM	DS	1	234)			
	END					

Screen clipping taken: 1/19/2022 2:22 PM

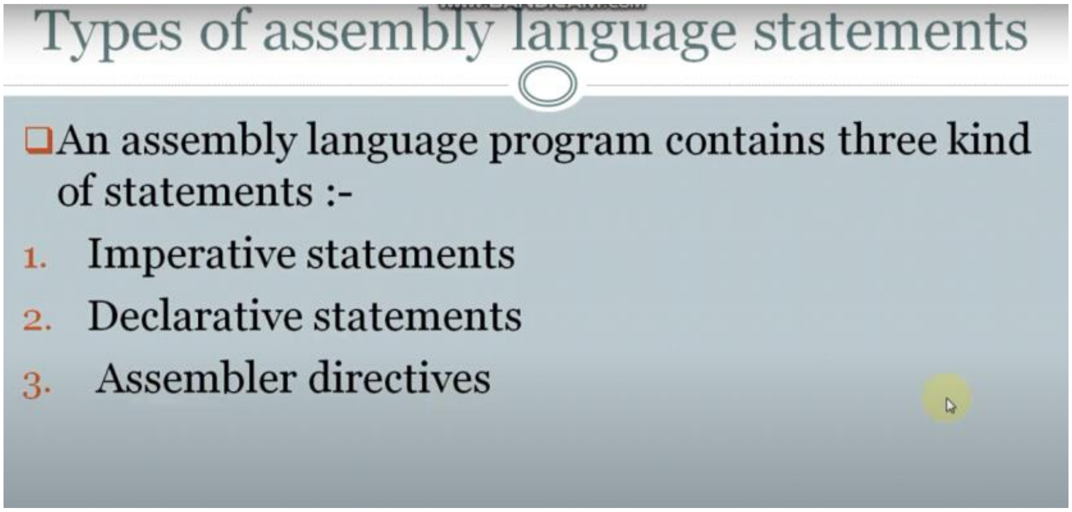


Types of assembly language statements

- An assembly language program contains three kind of statements :-

1. Imperative statements
2. Declarative statements
3. Assembler directives

Screen clipping taken: 1/19/2022 2:25 PM

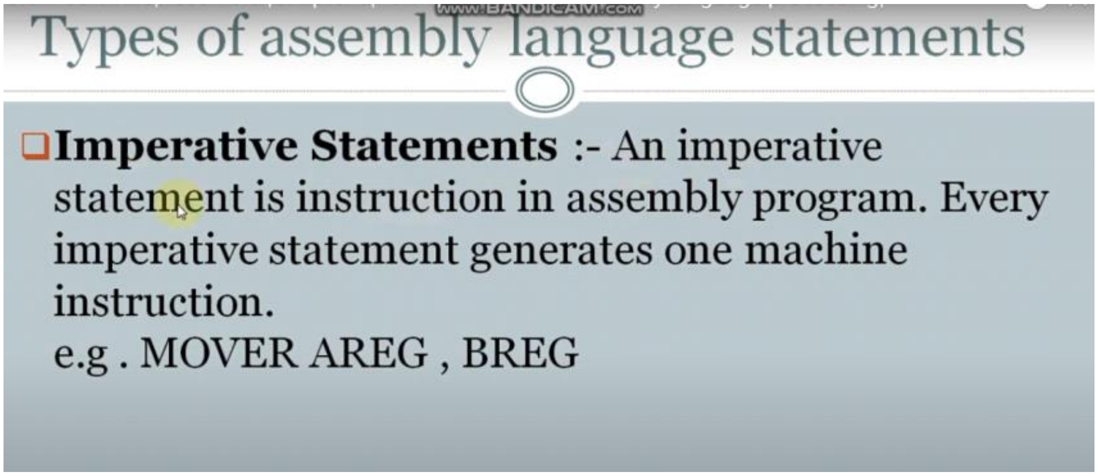


Types of assembly language statements

- An assembly language program contains three kind of statements :-

1. Imperative statements
2. Declarative statements
3. Assembler directives

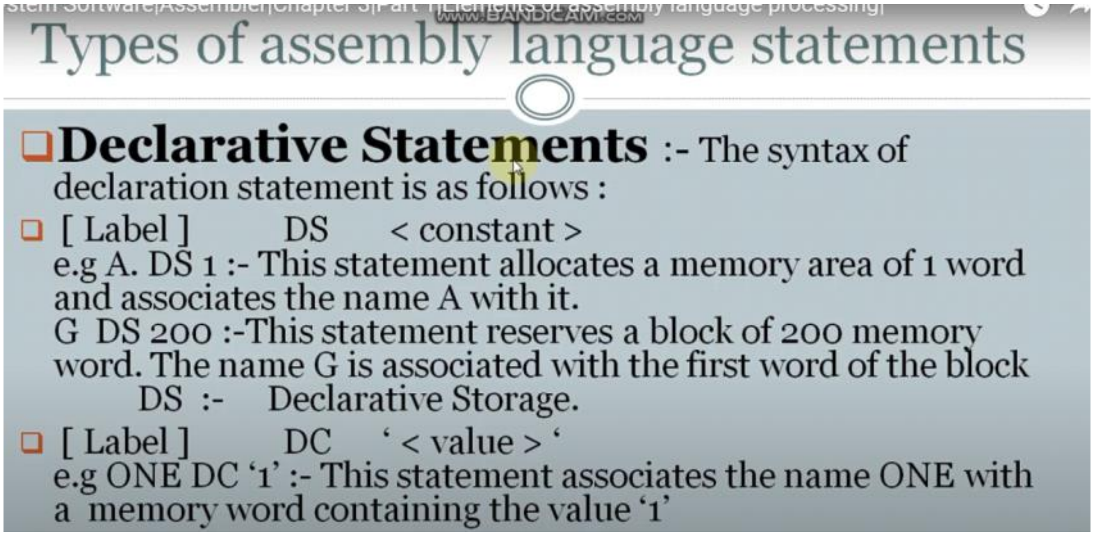
Screen clipping taken: 1/19/2022 2:25 PM



Types of assembly language statements

- **Imperative Statements** :- An imperative statement is instruction in assembly program. Every imperative statement generates one machine instruction.
  e.g . MOVER AREG , BREG

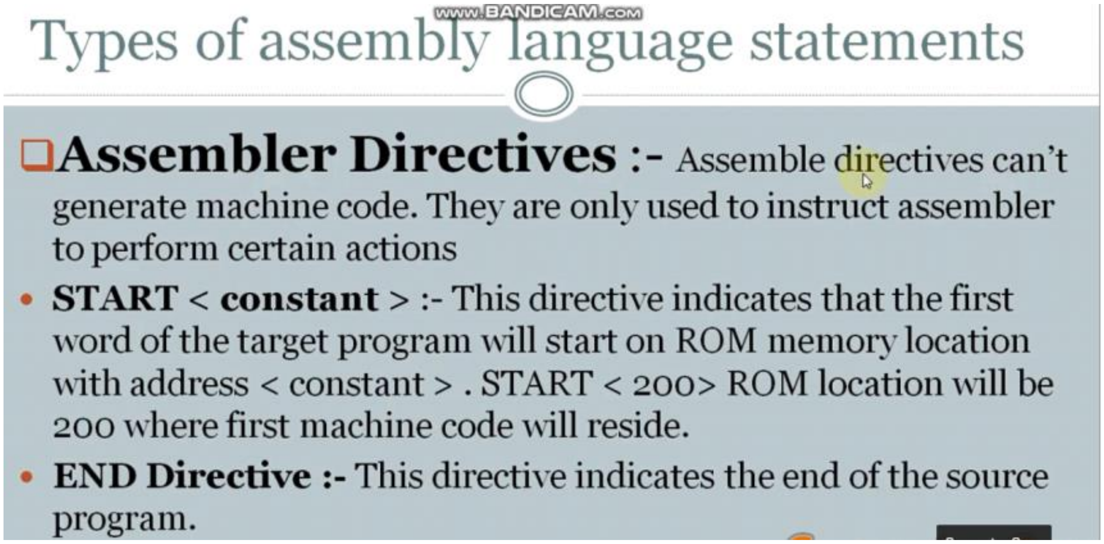
Screen clipping taken: 1/19/2022 2:26 PM



Types of assembly language statements

- **Declarative Statements** :- The syntax of declaration statement is as follows :
- [Label] DS < constant >
  e.g A. DS 1 :- This statement allocates a memory area of 1 word and associates the name A with it.
  G DS 200 :-This statement reserves a block of 200 memory word. The name G is associated with the first word of the block
  DS :- Declarative Storage.
- [Label] DC ' < value > '
  e.g ONE DC '1' :- This statement associates the name ONE with a memory word containing the value '1'

Screen clipping taken: 1/19/2022 2:27 PM



Types of assembly language statements

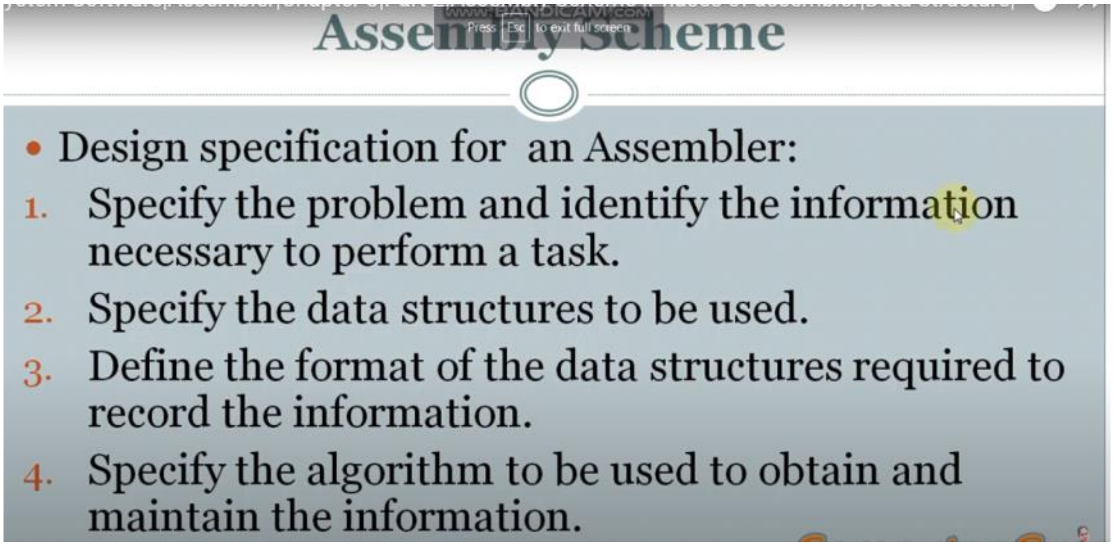
- **Assembler Directives :-** Assemble directives can't generate machine code. They are only used to instruct assembler to perform certain actions
- **START** < **constant** > :- This directive indicates that the first word of the target program will start on ROM memory location with address < constant > . START < 200> ROM location will be 200 where first machine code will reside.
- **END Directive :-** This directive indicates the end of the source program.

Screen clipping taken: 1/19/2022 2:29 PM

SYSTEM SOFTWARE

❑ASSEMBLY SCHEME

Screen clipping taken: 1/19/2022 2:38 PM



Assembly Scheme

- Design specification for an Assembler:

1. Specify the problem and identify the information necessary to perform a task.
2. Specify the data structures to be used.
3. Define the format of the data structures required to record the information.
4. Specify the algorithm to be used to obtain and maintain the information.

Screen clipping taken: 1/19/2022 2:39 PM

Assembler

Monday, January 24, 2022 10:27 AM

Assemblers

Assembler: Definition

- Translating source code written in assembly language to object code.

Assembly language source program

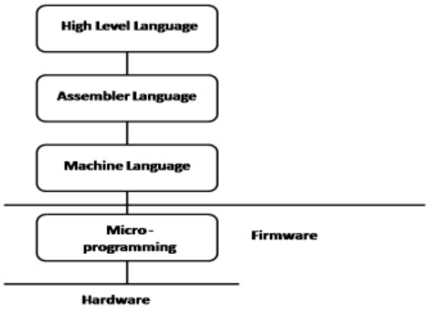
machine code

Source Program
- Mnemonic opcode
- Symbol

Assembler

Object code

Language Levels



High Level Language

Assembler Language

Machine Language

Micro-programming

Firmware

Hardware

Machine code

- Machine code:
  - Set of commands directly executable via CPU
  - Commands in numeric code
  - Lowest semantic level

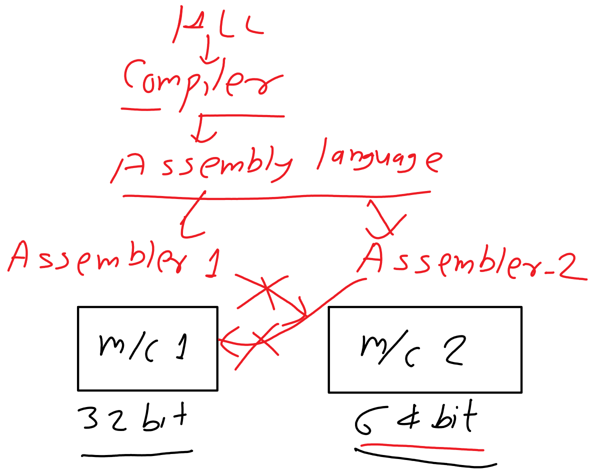
machine code will be in the form of 0's & 1's
ie. in binary form

Machine code language

- Structure:
  - Operation code
    - Defining executable operation
  - Operand address
    - Specification of operands
      - Constants/register addresses/storage addresses

OpCode	OpAddress

2/1/2022 11:00 AM



Elements of the Assembly Language Programming

- An Assembly language is a
  - machine dependent,
  - low level Programming language specific to a certain computer system.

Three features when compared with machine language are

1. Mnemonic Operation Codes ✓
2. Symbolic operands ✓
3. Data declarations ✓

ADD → Addition → operands

0011

Mnemonics – letter pattern/ association → help me remembering something

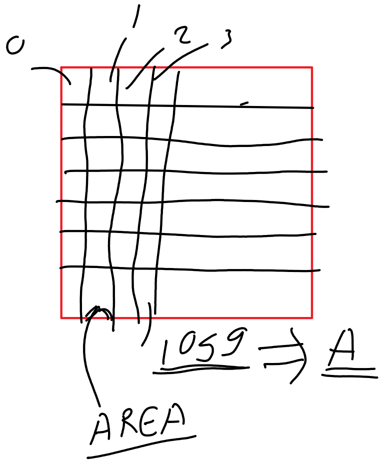
स्मृतिसहायक

Elements of the Assembly Language Programming

Mnemonic operation codes: eliminates the need to memorize numeric operation codes.

Symbolic operands: Symbolic names can be associated with data or instructions. Symbolic names can be used as operands in assembly statements (need not know details of memory bindings).

Data declarations: Data can be declared in a variety of notations, including the decimal notation (avoids conversion of constants into their internal representation).



Assembly language-structure

<Label>	<Mnemonic>	<Operand>	Comments

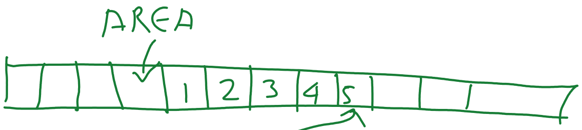
- Label
  - symbolic labeling of an assembler address (command address at Machine level)
- Mnemonic
  - Symbolic description of an operation
- Operands

AREA

Assembly language-structure

<Label>	<Mnemomic>	<Operand>	Comments

- Label
  - symbolic labeling of an assembler address (command address at Machine level)
- Mnemomic
  - Symbolic description of an operation
- Operands
  - Contains of variables or addresse if necessary
- Comments



AREA+5 → location of memory word which is 5 words away from AREA

Statement format

An Assembly language statement has following format:

[Label] *<opcode> <operand spec>[,<operand spec>..]*

If a label is specified in a statement, it is associated as a symbolic name with the memory word generated for the statement.

<operand spec> has the following syntax:

<symbolic name> **[+*<displacement>*] [(*<index register>*)]**

Eg. AREA, AREA+5, AREA(4), AREA+5(4)

AREA (4) → AREA address + value/Address in 4th index register

index register number

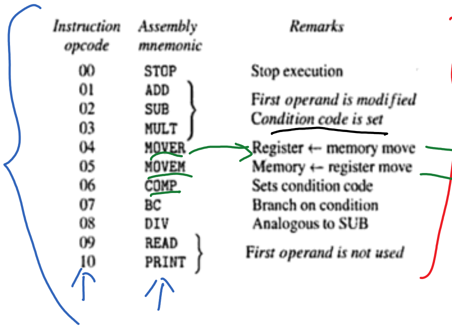
index register 4 → 50

AREA → 1024

AREA (4) ⇒ 1024 + 50 = 1074

Mnemonic Operation Codes

- Each statement has two operands, first operand is always a register and second operand refers to a memory word using a symbolic name and optional displacement.

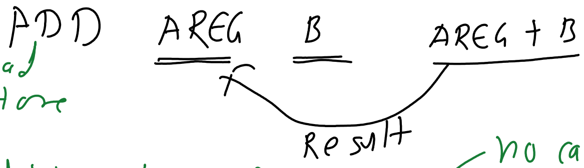


Instruction opcode	*Assembly mnemonic*	*Remarks*
00	STOP	Stop execution
01	ADD	*First operand is modified* *Condition code is set*
02	SUB	
03	MULT	
04	MOVER	Register ← memory move
05	MOVEM	Memory ← register move
06	COMP	Sets condition code
07	BC	Branch on condition
08	DIV	Analogous to SUB
09	READ	*First operand is not used*
10	PRINT	

→ Load

→ Store

A very simple hypothetical m/c



Condition code → info about → no carry / → Overflow / result <0 =0 >0

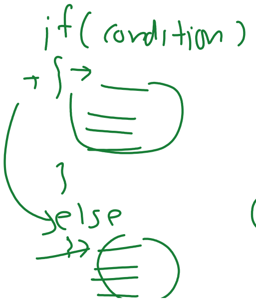
→ flags

Operation Codes

- ***MOVE*** instructions move a value between a memory word and a register
- ***MOVER*** – First operand is target and second operand is source
- ***MOVEM*** – first operand is source, second is target
- All arithmetic is performed in a register (replaces the contents of a register) and sets *condition code.*
- A Comparision instruction sets *condition code* analogous to arithmetics, i.e. without affecting values of operands.
- *condition code* can be tested by a Branch on Condition (BC) instruction and the format is:

  BC *<condition code spec>* , *<memory address>*

jump



$\frac{25}{4}$

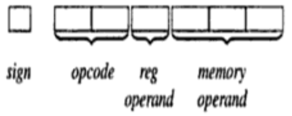
(25-4)-4-4-4

jump else (25-9-4-

- *condition code* can be tested by a Branch on Condition (BC) instruction and the format is:

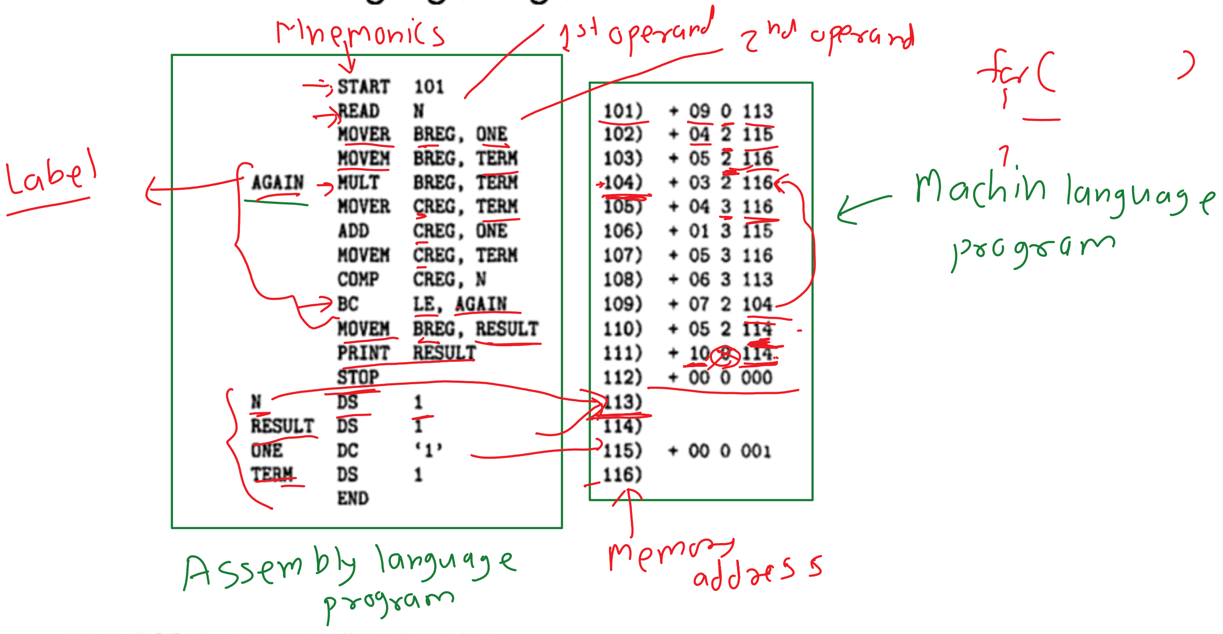
  BC *<condition code spec>* , *<memory address>*

Machine Instruction Format



- sign is not a part of the instruction
- Opcode: 2 digits, Register Operand: 1 digit, Memory Operand: 3 digits
- Condition code specified in a BC statement is encoded into the first operand using the codes 1- 6 for specifications LT, LE, EQ, GT, GE and ANY respectively
- In a Machine Language Program, all addresses and constants are shown in decimal as shown in the next slide

Example: ALP and its equivalent Machine Language Program



```
          START   101
          READ    N
          MOVER   BREG, ONE
          MOVEM   BREG, TERM
AGAIN     MULT    BREG, TERM
          MOVER   CREG, TERM
          ADD     CREG, ONE
          MOVEM   CREG, TERM
          COMP    CREG, N
          BC      LE, AGAIN
          MOVEM   BREG, RESULT
          PRINT   RESULT
          STOP
N         DS      1
RESULT    DS      1
ONE       DC      '1'
TERM      DS      1
          END
```

```
101)  + 09 0 113
102)  + 04 2 115
103)  + 05 2 116
104)  + 03 2 116
105)  + 04 3 116
106)  + 01 3 115
107)  + 05 3 116
108)  + 06 3 113
109)  + 07 2 104
110)  + 05 2 114
111)  + 10 0 114
112)  + 00 0 000
113)
114)
115)  + 00 0 001
116)
```

Assembly Language Statements

- An assembly program contains three kinds of statements:
  1) Imperative Statements ✓
  2) Declaration Statements ✓
  3) Assembler Directives ✓

Imperative Statements: They indicate an action to be performed during the execution of an assembled program. Each imperative statement is translated into one machine instruction.

ADD AREG, ONE

Assembly Language Statements

64 bit m/c

Assembly Language Statements

- **Declaration Statements:** syntax is as follows:

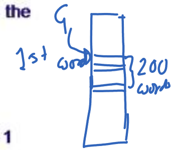
1 word → 64 bit m/c
→ 32 bit
16 bit

[Label] DS <*constant*>
[Label] DC '<*value*>'

- The DS (declare storage) statement reserves memory and associates names with them.
- Ex:

How many words to reserve – 1 word – How much storage → m/c dependent

A DS 1 ; reserves a memory area of 1 word, associating the name A to it
G DS 200 ; reserves a block of 200 words and the name G is associated with the first word of the block (G+6 etc. to access the other words)



- The DC (declare constant) statement constructs memory words containing constants.
- Ex:

ONE DC '1' ; associates name one with a memory word containing value 1

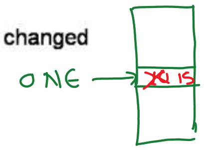
Assembly Language Statements

PI = 3.14 → Constant

Use of Constants

- The DC statement does not really implement constants
- it just initializes memory words to given values.
- The values are not protected by the assembler and can be changed by moving a new value into the memory word.
- In the above example, the value of ONE can be changed by executing an instruction

MOVEM BREG, ONE
15



Assembly Language Statements

Use of Constants

- An Assembly Program can use constants just like HLL, in two ways – as immediate operands, and as literals.
- 1) Immediate operands can be used in an assembly statement only if the architecture of the target machine includes the necessary features.
  - Ex: ADD AREG,5 → Constant
  - This is translated into an instruction from two operands – AREG and the value '5' as an immediate operand

Assembly Language Statements

Use of Constants

- 2) A ***literal*** is an operand with the syntax = '<value>'.
- It differs from a constant because its location cannot be specified in the assembly program.
- Its value does not change during the execution of the program

- It differs from a constant because its location cannot be specified in the assembly program.
- Its value does not change during the execution of the program.
- It differs from an immediate operand because no architectural provision is needed to support its use.

ADD AREG, ='5' →		ADD AREG, FIVE FIVE DC '5'
Use of literals	vs.	Use of DC

Assembly Language Statements

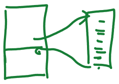
Assembler Directive

- Assembler directives instruct the assembler to perform certain actions during the assembly of a program.
- Some assembler directives are described in the following:

  1) START <*constant*> → location / Memory Area

- This directive indicates that the first word of the target program generated by the assembler should be placed in the memory word having address <*constant*>.

  2) END [<operand spec>] ← optional



- This directive indicates the end of the of the source program. The optional <operand spec> indicates the address of the instruction where the execution of the program should begin.

Advantages of Assembly Language

- The primary advantages of assembly language programming over machine language programming are due to the use of symbolic operand specifications. ONE, PI ...

  (in comparison to machine language program) → use addresses

- Assembly language programming holds an edge over HLL programming in situations where it is desirable to use architectural features of a computer. → Some Architectures supports some special instruction which can speed up execution

  (in comparison to high level language program)

Fundamentals of LP → Language Processing

- Language processing = analysis of source program + synthesis of target program
- Analysis of source program is specification of the source program
  - Lexical rules: formation of valid lexical units(tokens) in the source language
  - Syntax rules : formation of valid statements in the source language
  - Semantic rules: associate meaning with valid statements of the language

operators

Percent_profit := (profit * 100) / cost_price ;

identifiers

constant

Fundamentals of LP

- Synthesis of target program is construction of target language statements
  - – Memory allocation : generation of data structures in the target program
  - – Code generation

A simple Assembly Scheme

- There are two phases in specifying an assembler:

1. Analysis Phase
2. Synthesis Phase(the fundamental information requirements will arise in this phase)

A simple Assembly Scheme

Design Specification of an assembler

There are four steps involved to design the specification of an assembler:

1) Identify information necessary to perform a task.
2) Design a suitable data structure to record info. ✓
3) Determine processing necessary to obtain and maintain the info.
4) Determine processing necessary to perform the task

Making a tea ✓

Synthesis Phase: Example

Consider the following statement:

MOVER BREG, ONE

Move value at address pointed by ONE to B regeister

✓ The following info is needed to synthesize machine instruction for this stmt:

1. Address of the memory word with which name **ONE** is associated [depends on the source program, hence made available by the Analysis phase].
2. Machine operation code corresponding to **MOVER** [does not depend on the source program but depends on the assembly language, hence synthesis phase can determine this information for itself]

Note: Based on above discussion, the two data structures required during the synthesis phase are described next

Data structures in synthesis phase

1) **Symbol Table** –*built by the analysis phase* ✓

Symbol table _ Analysis

Data structures in synthesis phase

Symbol table _ Analysis

1) **Symbol Table** *–built by the analysis phase* ✓
  - The two primary fields are name and address of the symbol used to specify a value.

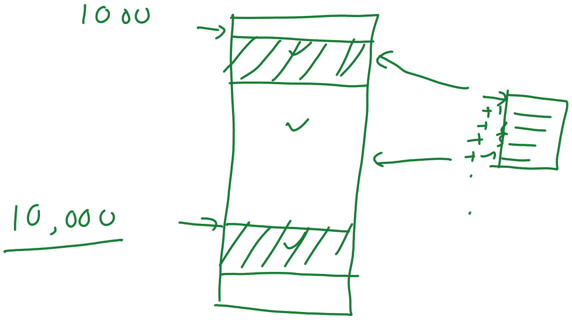
Mnemonics Table *–already present*
  - The two primary fields are *mnemonic* and *opcode,* along with *length.*

Synthesis phase uses these tables to obtain
  - The machine address with which a name is associated. ✓
  - The machine op code corresponding to a mnemonic. ✓

- The tables have to be searched with the
  - Symbol name and the mnemonic as keys

Analysis Phase → to build symbol table

- Primary function of the Analysis phase is to build the symbol table.
  - It must determine the addresses with which the symbolic names used in a program are associated
  - It is possible to determine some addresses directly like the address of first instruction in the program (ie.,start) →
  - Other addresses must be inferred
  - To determine the addresses of the symbolic names we need to fix the addresses of all program elements preceding it through *Memory Allocation*.
- To implement *memory allocation* a data structure called *location counter* is introduced. [LC]



Analysis Phase – Implementing memory allocation

- ✓ LC(location counter) :
  - is always made to contain the address of the next memory word in the target program.
  - It is initialized to the constant specified at the START statement.
- When a LABEL is encountered,
  - it enters the LABEL and the contents of LC in a new entry of the symbol table.

    LABEL – e.g. N, AGAIN, SUM etc
  - It then finds the number of memory words required by the assembly statement and updates the LC contents
- To update the contents of the LC, analysis phase needs to know lengths of the different instructions
  - This information is available in the Mnemonics table and is extended with a field called length
- We refer the processing involved in maintaining the LC as LC Processing

N → 4 words

LC - 1000

LC = LC+4
=> = 1004

{ 1001
1002
1003

Example

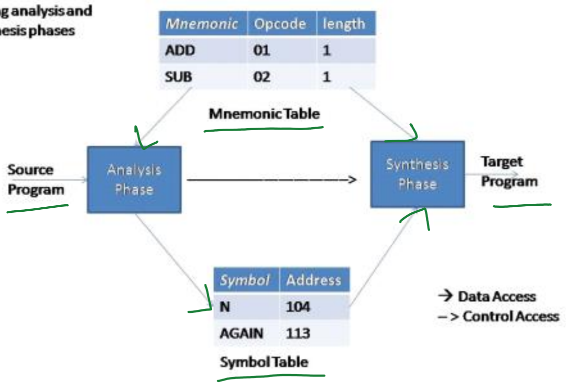
```
START 100
MOVER  BREG, N          LC = 100    (1 byte)
MULT  BREG, N           LC = 101    (1 byte)
STOP                    LC = 102    (1 byte)
N  DS  5                LC = 103
```

Symbol	Address
N	103

- Since there the instructions take different amount of memory, it is also stored in the mnemonic table in the "length" field

Mnemonic	Opcode	Length
MOVER	04	1
MULT	03	1

Data structures of an assembler
During analysis and
Synthesis phases



Mnemonic	Opcode	length
ADD	01	1
SUB	02	1

Symbol	Address
N	104
AGAIN	113

Data structures

- Mnemonics table is a fixed table which is merely accessed by the analysis and synthesis phases
- Symbol table is constructed during analysis and used during synthesis

VIMP

Tasks Performed : Analysis Phase

Tasks Performed : Analysis Phase

- ① Isolate the labels, mnemonic, opcode and operand fields of a statement.
- ② If a label is present, enter (symbol, <LC>) into the symbol table.
- ③ Check validity of the mnemonic opcode using mnemonics table. → Prebuilt
- ④ Update value of LC.

Tasks Performed : Synthesis Phase ✓

- Obtain machine opcode corresponding to the mnemonic from the mnemonic table.
- obtain address of the memory operand from symbol table.
- Synthesize a machine instruction or machine form of a constant, depending on the instruction.

Mnemonic table ✓
Opcode → M/c code

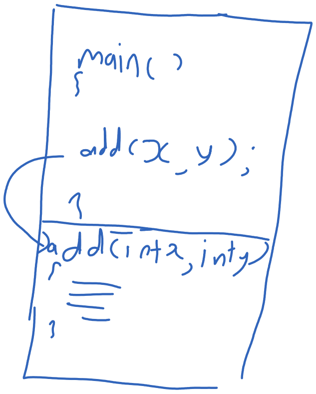
Assembler's functions

- ① Convert mnemonic operation codes to their machine language equivalents
- ② Convert symbolic operands to their equivalent machine addresses
- ③ Build the machine instructions in the proper format
- ④ Convert the data constants to internal machine representations
- ⑤ Write the object program and the assembly listing

Assembly language program ⇒ object code
m/c language program

Assembler:Design

- The design of assembler can be of:
  - Scanning (tokenizing)
  - Parsing (validating the instructions)
  - Creating the symbol table ✓
  - Resolving the forward references
  - Converting into the machine language ✓



Assembler Design

- Pass of a language processor – one complete scan of the source program

Pass – 1 single scan/reading of the

Assembler Design

- Pass of a language processor – one complete scan of the source program
- Assembler Design can be done in:
  - Single pass
  - Two pass
- Single Pass Assembler:
  - Does everything in single pass
  - Problem → Cannot resolve the forward referencing
- Two pass assembler:
  - Does the work in two pass ✓
  - Resolves the forward references ✓

Pass – 1 single scan/reading of the source program

Difficulties: Forward Reference

- Forward reference: reference to a label that is defined later in the program.

Loc	Label	Operator	Operand	
1000	FIRST	STL	RETADR	
1003	CLOOP	JSUB	RDREC	
...	...	...	...	...
1012		J	CLOOP	
...	...	...	...	...
1033	RETADR RESW	1		

Backpatching

→ used to handle/resolve the problem of forward reference

- The problem of forward references is handled using a process called backpatching
  - Initially, the operand field of an instruction containing a forward reference is left blank
  - Ex: MOVER BREG, ONE can be only partially synthesized since ONE is a forward reference
  - The instruction opcode and address of BREG will be assembled to reside in location 101
  - To insert the second operand's address later, an entry is added as Table of Incomplete Instructions (TII)
  - The entry TII is a pair (<instruction address>, <symbol>) which is (101, ONE) here

data structure

TII - Table of Incomplete Instruction

Instruction address	Symbol
101	ONE

Backpatching

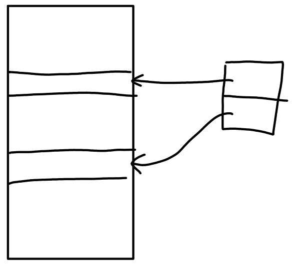
- The problem of forward references is handled using a process called backpatching
  - When END statement is processed, the symbol table would contain the addresses of all symbols defined in the source program
  - So TII would contain information of all forward references
  - Now each entry in TII is processed to complete the instruction
  - Ex: the entry (101, ONE) would be processed by obtaining the address of ONE from symbol table and inserting it in the operand field of the instruction with assembled address 101.
  - Alternatively, when definition of some symbol L is encountered, all forward references to L can be processed

END- Assembler directive

Advanced Assembler Directives ✓

- 1. ORIGIN
  - This directive is like START instruction, which indicates address of the next consecutive instruction or data.
  - Format of this statement is as follows
  - ORIGIN <address spec>
  - <address spec> may be operand or constant, symbol or symbolic expression.
  - This directive indicates that LC should be set to The address given by <address spec>
  - The ORIGIN directive is useful when the machine code is not stored in consecutive memory location.
  - ORIGIN provides ability to perform LC processing in relative manner rather than absolute manner

LC - Location Counter



Advanced Assembler Directives

- 1. ORIGIN
- ORIGIN in Relative manner ← LOOP-202
- ORIGIN LOOP +2 Loop +2 = (202+2 = 204)
- MULT CREG, B → 204
- here LC at LOOP is 202, than now LC will set to location 204 and the address of machine code for MULT CREG,B will become 204
- The statement LAST+1 sets LC to location 217
- Equivalent effect can be achieved by using statement ORIGIN 204 and ORIGIN 217, however absolute addresses used in these statements would needed be changed if the address specification of START statement is changed.

Sr. no.	Assembly program	LC
1	START 100	
2	LOOP MOVER BREG='2'	100
3	MOVER AREG,N	101
4	ADD AREG='1'	102
5	ORIGIN LOOP	
6	NEXT BC ANY,LOOP	100

Need to find out address of symbol associated with "LOOP"

Advanced Assembler Directives

- 2. EQU
  - <symbol> EQU <address spec>
  - Ex: A EQU B
  - Address of B is assigned to A in symbol table.
  - This directive simply associate the name <symbol> with < address spec>.
  - where <address spec> may be constant or operand.
  - The EQU statement is defers from the DC/DS statement as no LC processing is implied

Advanced Assembler Directives

- 2. LTORG

Advanced Assembler Directives

- 2. LTORG
  - LTORG

    '=5'

    '=1'

    Literals
  - This directive allocates memory to all literals of current pool and update literal table, pool table
  - Format of this instruction is as follows
  - LTORG.
  - If LTORG statement is not present, literals are placed after the END statement.

```
1            START    200
2            MOVER    AREG, ='5'        200)   +04 1 211
3            MOVEM    AREG, A           201)   +05 1 217
4   LOOP     MOVER    AREG, A           202)   +04 1 217
5            MOVER    CREG, B           203)   +05 3 218
6            ADD      CREG, ='1'        204)   +01 3 212
7            ...

12           BC       ANY, NEXT         210)   +07 6 214
13           LTORG
                      ='5'              211)   +00 0 005
                      ='1'              212)   +00 0 001
14           ...
15  NEXT     SUB      AREG, ='1'        214)   +02 1 219
16           BC       LT, BACK          215)   +07 1 202
17  LAST     STOP                       216)   +00 0 000
18           ORIGIN   LOOP+2
19           MULT     CREG, B           204)   +03 3 218
20           ORIGIN   LAST+1
21  A        DS       1                 217)
22  BACK     EQU      LOOP
23  B        DS       1                 218)
24           END
25                    ='1'              219)   +00 0 001
```

ASSEMBLY PROGRAM ILLUSTRATING ORIGIN AND LTORG

- The LTORG statement ~~permits programmer~~ to specify where literal should be placed. by default assembler places literals after end statement
- At Every LTORG statement, as also at END statement The assembler allocates memory to the literals of the literal pool. The pool contains all literals used in the program since start of program or start of LTORG statement.
- in Program of previous slide , literals '=5' and '=1' are added to literal pool with addresses 211 and 212
- A new literal pool now started and value '=1' is put in to this pool in statement 15. this value is allocated at address 219 of second pool of literals rather than location 213 of first pool

Assembler Design

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

Assembler Design

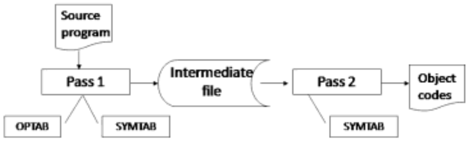
- Assembler directives are pseudo instructions.
  - They provide instructions to the assemblers itself.
  - They are not translated into machine operation codes.

Assembler Design

- First pass:
  - Scan the code by separating the symbol, mnemonic op code and operand fields
  - Build the symbol table
  - Perform LC processing
  - Construct intermediate representation
- Second Pass:
  - Solves forward references
  - Converts the code to the machine code

Two Pass Assembler

- Read from input line
  - LABEL, OPCODE, OPERAND


Source program
Pass 1
Intermediate file
Pass 2
Object codes
OPTAB
SYMTAB
SYMTAB

Data Structures in Pass I

- OPTAB – a table of mnemonic op codes
  - Contains mnemonic op code, class and mnemonic info
  - Class field indicates whether the op code corresponds to
    - an imperative statement (IS),
    - a declaration statement (DL) or
    - an assembler Directive (AD)
  - For IS, mnemonic info field contains the pair (machine opcode, instruction length)
  - Else, it contains the id of the routine to handle the declaration or a directive statement
  - The routine processes the operand field of the statement to determine the amount of memory required and updates LC and the SYMTAB entry of the symbol defined

Data Structures in Pass I

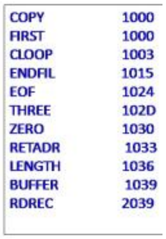
- SYMTAB - Symbol Table
  - Contains address and length
- LOCCTR - Location Counter
- LITTAB – a table of literals used in the program
  - Contains literal and address
  - Literals are allocated addresses starting with the current value in LC and LC is incremented, appropriately

OPTAB (operation code table)

- Content
  - Menmonic opcode, class and mnemonic info
- Characteristic
  - static table
- Implementation
  - array or hash table, easy for search

SYMTAB (symbol table)

- Content
  - label name, value, flag, (type, length) etc.
- Characteristic
  - dynamic table (insert, delete, search)
- Implementation
  - hash table, non-random keys, hashing function



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