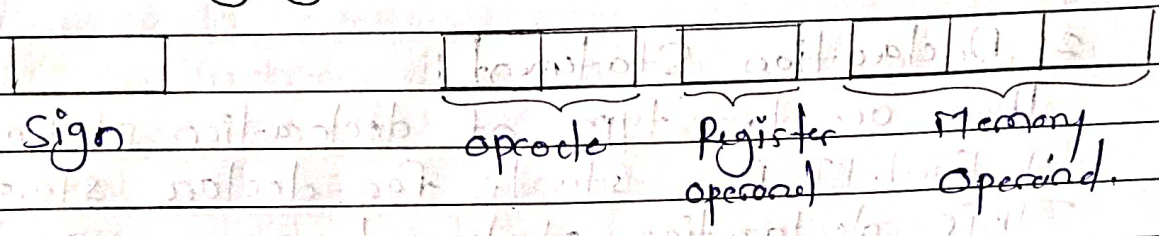
D D M M Y Y Y Y

Assignment No. 2.

1) Explain Assembly language statement format and machine instruction format.

Machine language instruction format :-



Above figure shows the format of machine instruction. It consist of opcode, Register operand, and memory operand.
Opcode occupy 2 digits. Register operand occupy 1 digit and memory operand occupy 3 digits.
The sign is the not part of instruction.
The condition code is specified in BC statement.
The opcode defines the particular instruction code used in program
Register operand defines which particular register is involved in operation. and memory operand defines at which particular location of memory address it is ~~sorted~~ stored,

Assembly language Statement.

Assembly language has three kinds of statements
1) Imporative statement.
2) Declaration statement.
3) Assembler Directives.

1) Imparative statement :
An imporative statement indicates an action to be performed during the execution of progrm for ex. An arithmatic operation.
Each imporative statement translates into one machine instruction.

2) Declaration Statement :
There are two types of declaration statements.
1) [label] DS - Stands for declare storage.
This declaration statement reserves an area of memory an associate a symbolic name with it.
Consider following DS statement
A DS 1
This statement reserves a memory area of one word with associate with name 'A'.
G DS 200
This statement reserves a block of 200 memory word an associate name G with first word of block.

2) [label] DC - Stands for Declare constant.
This statement construct memory word containing constants.
The statement
ONE DS '1'
associates the name ONE with a memory word containing value 1.

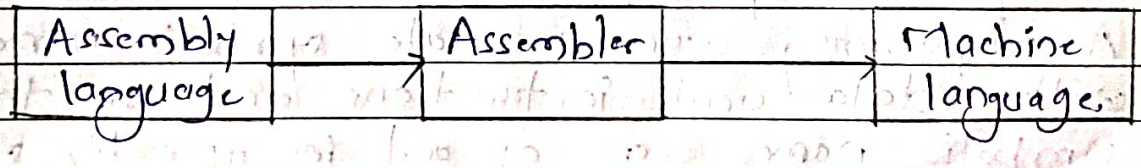
3) Assembler Directive:
Assembler Directive instructs the assembler to perform certain action while assembling a program. There are two types of assembler directive.

1) START <constant>
The start directive construct assembler to place first word of target program generated by it in the memory word having address constant.

2) END [<operand specification>]
The end directive indicates the end of source program. The optional <operand specification> indicates that execution of program should begin with instruction whose address is specified by operand specification.

Q.2) Explain elements of assembly language.

Assembler is one of the system software which is used to convert assembly language into machine language.



Elements of assembly language:
The machine language of computer has numeric instructions that take operands which are either address of CPU register or memory location or

small number in binary notation used in immediate operands.
An assembly language is machine dependent low level programming language that is specific to a certain computer or certain family of computers.
Each statement in assembly language program corresponds to instructions in computer or it is a declaration, statement or directive to assembler.
Assembly language provides following 3 basic properties that simplify programming.

1) Mnemonic Operation Code:
Mnemonic operation code for machine language also called as mnemonic opcode are easier to remember and useful than mnemonic operation code. Their use also enables the assembler to detect invalid operation codes in a program.
Ex. MUL, SUB, ADD

2) Symbolic Operand:
A programming can associate symbolic names with data and instructions and use this symbolic names as operand in assembly statement. This facility frees the programmer from having to think of numeric addresses in a program thereby changes described earlier. We use term symbolic name only in a formal contexts.

elsewhere we simply say name.

3) Data Declaration:
Data can be declared in a variety of notations including the decimal notation. It avoids the need to manually specify constants in representation that a computer can understand for example.

NUM
NUM

Q.3) Explain Pass Structure of Assembler.
Pass of language processor performs language processing functions on every statement in a source program or in its equivalent representation code.

Assembler generates instruction of symbol or mnemonics in one operation field & find the values of symbol to produce machine code. If assembler does this work in one scan or pass then it is called as single pass assembler.

1) Single Pass Assembler:
A single pass assembler defines symbols & literals and separates the symbol, mnemonic opcode & operand field.

Single pass assembler will perform analysis. It builds the symbol table & literal table to remember

them respectively.

For keeping track of location counter (LC) & performs processing.

Processes & construct the intermediate representation.

2) Two Pass Assembler:

Passes of two pass assembler performs following tasks:

Pass I: 1) Separates the symbol, mnemonic opcode & operand fields.

2) Build the symbol table.

3) Perform LC Processing.

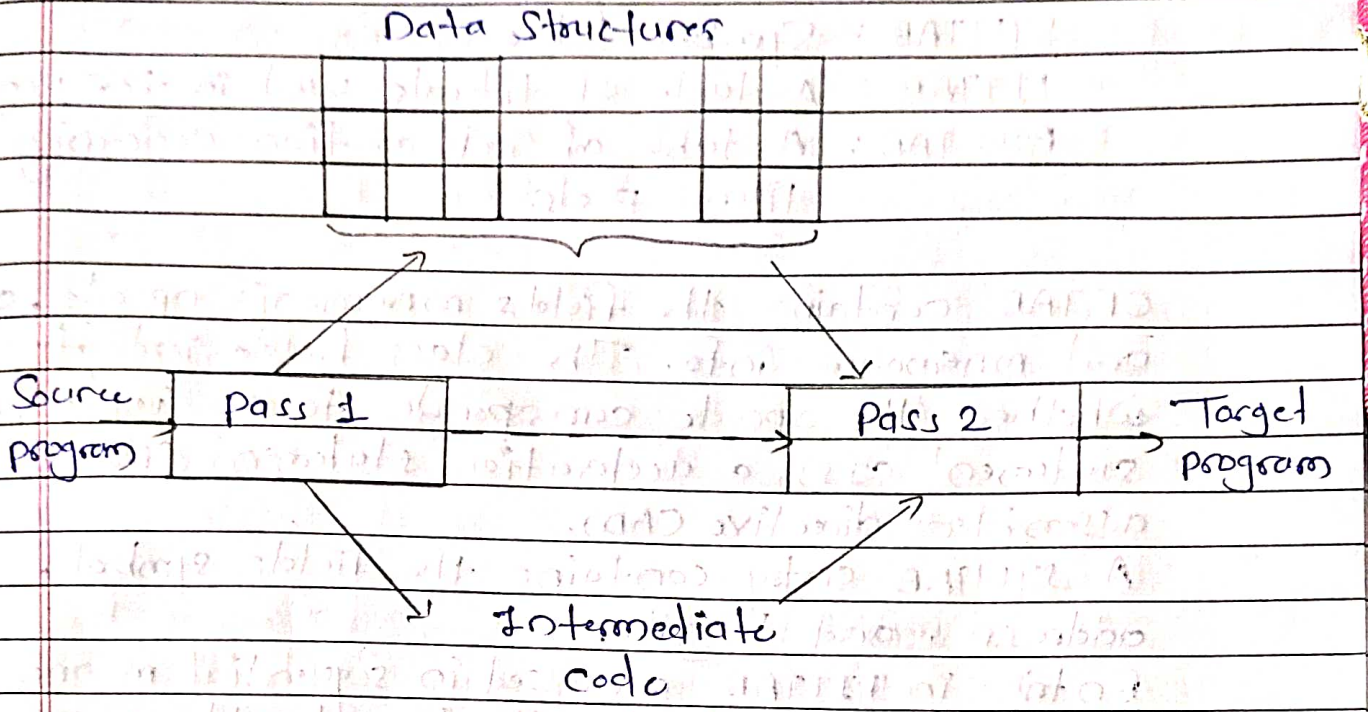
4) Construct intermediate representation.

Pass II: Synthesis of target program.

Pass I performs analysis of target program & synthesis of intermediate representation & pass-II performs / processes intermediate representation to synthesizes the target program.

Two pass assembler forwards reference easily.

Following diagram shows the working of two pass assembler.



Intermediate representation consist of two main components data structure & Intermediate code. First pass performs the synthesis of target program. This organization handles the forward reference to a symbol naturally because the address of each symbol would be known before program synthesis begins.

(Q.4) Explain one pass assembler.

Pass 1 uses the following data structures.

OPTAB : A table of mnemonic opcodes and related information.

SYMTAB : Symbol table
LITTAB : A table of literals used in the program
POOLTAB : A table of information concerning literal pools.

OPTAB contains the fields mnemonic opcode, class and mnemonic info. The class field indicates whether the opcode corresponds to an imperative statement (IS), a declaration statement (DL) or an assembler directive (AD).
A SYMTAB entry contains the fields symbol, address, and length.
Entries in LITTAB are used in sequential manner. It contains the fields literal and address.
An entry in POOLTAB pertains to a pool of literals. It contains the single field literal no. to indicate which entry in the LITTAB contains the 1st literal of the pool.

Processing of an assembly statement begins with the processing of its label field. If it contains a symbol, the symbol and the address contained in location counter is copied into a new entry of SYMTAB.
Then, the functioning of Pass I centers around the interpretation of the OPTAB entry for the mnemonic used in the statement.

mnemonic opcode	class	mnemonic info
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	⋮	

OPTAB

Symbol	address	length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

SYMTAB

	Value	address
1	='5'	
2	='1'	
3	='1'	

LITTAB

first	#literals
1	2
3	1
4	0

POOLTAB

fig. Data structures of assembler Pass-I.

The assembler uses the LITTAB and POOLTAB as follows:

At any stage, the current literal pool is the last pool in LITTAB.

On encountering an LTORG statement (or the END stmt), literals in the current pool are allocated addresses starting with the current address in the location counter and the address in the location counter is appropriately incremented.

Q.5) Explain Advanced Assembler Directives.

ORIGIN:

The syntax of this directive is

ORIGIN <address specification>

where <address specification> is an <operand specification> or <constant>.

This directive instructs the assembler to put the address given by <address specification> in the location counter.

The ORIGIN statement is useful when the target program does not consist of a single contiguous area of memory.

The ability to use an <operand specification> in the ORIGIN statement provides the ability to change the address in the location counter in a relative rather than absolute manner, which has benefits analogous.

EQU:

The EQU directive has the syntax,

<symbol> EQU <address specification>

where <address specification> is either a <constant> or <symbolic name> ± <displacement>.

The EQU statement simply associates the name <symbol> with the address specified by <address specification>. However, the address in the location counter is not affected.

LTORG:

The assembler should put the values of literals in such a place that control does not reach any of them during execution of the generated program.

The LTORG directive, which stands for 'origin for literals', allows a programmer to specify where literals should be placed.

The assembler uses the following scheme for placement of literals:

When the use of a literal is seen in a statement the assembler enters it into a literal pool unless a matching literal already exists in the pool.

At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of the literal pool and clears the literal pool.

This way, a literal pool would contain all literals used in the program since the start of the program or since the previous LTORG statement.

Thus, all references to literals are forward references by definition.

If a program does not use an LTORG statement, the assembler would enter all literals used in the program into a single pool and allocate memory to them when it encounters the END statement.

Q.6) What is IC Unit? Discuss two variants of interme code for any imperative statements.

IC Unit:

The intermediate code consists of a sequence of intermediate code units (IC units).

Each IC unit consist of the following three fields:

1) Address
2) Representation of the mnemonic opcode.
3) Representation of operands.

Address	Mnemonic opcode	Operands

fig. format of IC unit.

Variant forms of intermediate codes, specifically the operand and address fields, are used in practice due to the trade-off between processing efficiency and memory economy.

Mnemonic opcode field:

Declaration statements		Assembly directives	
DC	01	START	01
DS	02	END	02
		ORIGIN	03
		EQU	04
		LTORG	05

The mnemonic opcode field contains the pair of the form

(Statement class, code).

Intermediate Code for any Imperative Statements:
We consider two variants of intermediate code which differ in the information contained in their operand fields.

1) Variant I:
The 1st operand is represented by a single digit number which is a code for a register (#1-4 for AREG-DREG) or the condition code itself (1-6 for LT-ANY).
The 2nd operand which is memory operand, is represented by a pair of the form,

(operand class, code).

Where operand class is one of C, S, L standing for constant, symbol and Literal respectively.
e.g.
The operand descriptor for the statement

START 200 is (C, 200).

for a symbol or literal, code field contains the ordinal no. of the operands entry in SYMTAB or LITTAB.
Two kinds of entries may exist in SYMTAB at any time i.e. for defined symbols and for forward references.

2) Variant II:

This variant differ from variant I. In that the operand field of the intermediate code may be either in the processed form as in variant I, or in the source form itself.

for a declarative statement or an assembler directive the operand field has to be processed in the first pass to support LC processing.

Hence the operand field of its intermediate code would contain the processed form of the operand.

for imperative statements, the operand field is processed to identify literal reference and enter them in the LITTAB, hence operands that are literals are represented as (L, m) in the intermediate code.

Q.7) What are assembler directives? List out and explain any two assembler directives.

Assembler directives are the instructions used by the assembler at the time of assembling a source program.

More specifically, we can say, assembler directives are the commands or instructions that control the operation of the assembler

Assembler directives are the instructions provided to the assembler, not the processor as the processor has nothing to do with these instructions. This instructions are also known

as pseudo-instructions or pseudo-opcode.

What Assembler directives specifically do?

Assemble Directives:

1) Show the beginning and end of a program provided to the assembler.
2) Used to provide storage locations to data.
3) Used to give values to variables.
4) Define the start and end of different segments, procedures or macros etc. of a program.

We describe 2 directives START & END:

1) START <constant>:
START directive instructs the assembler to place the first word of target program generated by it in memory word using / having address <constant>.

2) END [<operand specification>]:
END directive indicates the end of source program the optimal <operand specification> indicates the execution of program should begin with instruction whose address is specified by operand specification.

Assembly statement

```
START
MOVER AREG PROF11
MULT AREG 100
```

DIV AREG, COST_PRICE
MOVEM AREG PERCENT_PROFIT