Chapter 03

Assembler machine Independent Features and Design Options

* Machine Independent features:
The features that does not depend on the architecture of machine are known as machine Independent features.

Some of the machine Independent features are:-

<i> assembler with LITERALS
<ii> Symbol defining Statement
<iii> Expressions
<iv> Program Blocks
<v> Control Sections

(1) LITERALS:

- Literals are the constants that are literally present in the Instruction.
- The programmers write the value of a constant operand as a part of the Instruction that uses it.
- This avoids the need to define the constant elsewhere in the program and make up a label for it.

- When we use literals, there is no need for programmer to define constant explicitly.
- A literal is defined / identified with the prefix =, followed by a specification of the literal value.

E.g: LDA =X'05'

In the above statement, literal X specifies a 1-byte literal with the hexadecimal value 05.

- Literal pools: All the literal operands that are used in a program are gathered together into literal pools.
- Usually, literal pools are present at the end of the program.
- It contains the assigned addresses and the generated data values.
- For some cases, the distance between the referring statement & the referred statement is expected to be low.
- So, it is desirable to place the literal pool or allocate the memory for literals at some other memory location.

- This is possible by using LTORG.
- LTORG is an assembler directive that creates a literal pool that contains all of the literal operands used since (from beginning) of the program.

② Symbol Defining Statements :-

- Some assemblers provide an assembler directive that allows the programmer to define symbols & specify their values.
- EQU - Using assembler directives like EQU, we can define new symbols.

E.g. MAXLEN EQU 2000

- Once, the symbols are defined.
- Such symbols are added into Symbol-table by assemblers.
- The syntax for using EQU is :-

symbol EQU value

- This statement defines the given symbol (i.e. enters it into SYMTAB) and assigns to it the value specified.

- One common use of EQU is to establish symbolic names that can be used for improved readability in place of numeric values.

E.g: +LDT #4096
// Here 4096 is the maximum length record that we could read with RDREC.

So, we can use :-
MAXLEN EQU 4096
+LDT MAXLEN

- Another common use of EQU is in defining mnemonic names for registers.

E.g:
A EQU 0
X EQU 1
L EQU 2
:

- ORG - Using this assembler directive, indirectly values can be assigned to symbols.

E.g If we use ORG 5000,
then all subsequent statements that we write are assigned addresses from 5000.

- It affects addresses of the symbol & labels.

* → gives the address of the next unassigned memory location.

(3) Expressions

- Assemblers allow the use of expressions within a program.
- Each expression is evaluated by the assembler to produce a single operand address or value.
- The expressions can be the arithmetic exp. formed according to the normal rules using the operators +, −, * and /.
- The '*' symbol gives the address of the next unassigned memory location.

E.g. 106 BUFEND EQU *

Here, BUFEND is assigned with the address of the next byte after the Buffer area.

- BUFEND marks the end of the buffer area.

There are 02 types of expressions based on the value they produce:-

(1) Absolute Expression

(2) Relative Expression

Absolute value - Constants

Relative value - labels, references to location counter.

I). Absolute Expressions:

- It is an expression that contains only absolute terms.

- These exp. might also consider relative terms provided that the relative terms occur in pair & with different signs.

> e.g: 107 MAXLEN EQU BUFFEND-BUFFER
> // In this exp, both BUFFEND and BUFFER are relative terms, but when they are subtracted, we get the length of buffer area in bytes which is a constant. So, it is an absolute expression.

- However, we cannot use relative terms with multiplication or division operation for absolute expressions.

II). Relative Expressions:

- It is an expression in which all the relative terms except one can be paired, and the remaining unpaired relative term must have a positive sign.

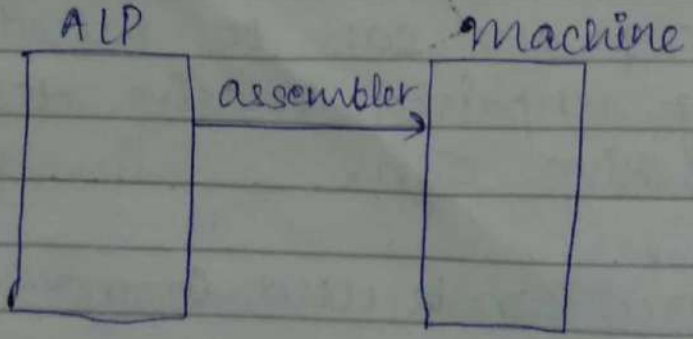
- No relative term can be used in multiplication & division operation.

The expressions that do not satisfy the conditions for either absolute or relative are flagged as errors.

e.g: BUFEND + BUFFER, 100 - BUFFER, 3 * BUFFER
//These expressions

(4) Program Blocks:

- The source program contains many blocks such as subroutines, data area, buffer area, etc.
- When the assembler converts this source program (ALP) into machine code, a single ~~blo~~ code is generated.
- The program is divided into several blocks/segments during translation.
- Then after translation, a single object program with different blocks is generated.



- Within the object program, the generated m/c instructions and data appear in the same order as they appear in the source program.

- Program Blocks mainly refer to the segments of code that are rearranged within a single object program unit.

(5) Control Sections:-

- Control Sections refer to segments that are translated into independent obj program units.

- In case of control sections, several object programs/codes are generated.

- All these blocks will be independent i.e. they can be loaded and used independently.

- One object code can call some other var which is defined in some other object code.

- Here, two assembler directives are used:-

<1> EXTDEF (External defined statements)
↳ It is used to define the variable which can be used in other control section.

30-1
=29

+3

<iii> EXTREF (External reference Statements)

↳ It is used to refer some variable in a block which is not defined in the same control section.

E.g EXTDEF BUFFER, BUFFEND
EXTREF RDREC

*** ASSEMBLER DESIGN OPTIONS ***

* One-Pass Assembler :-

- One-pass Assemblers scan the ALP only once to convert into a machine code.
- It tries to eliminate / resolve the issue of forward reference, since the assembler does not know what address to be inserted in the translated instruction at the time of forward jump.
- It resolves this issue by having all the data items defined at the starting of the program before the instructions begin.
- There are two main types of one-pass assembler:
  <i> load & go <ii> Other

FUNCTIONS

<i> Load & go:

- This type of assemblers generate object code directly in memory for immediate execution.
- Here, no object program is written out and no loader is needed.
- It is useful for a system that is oriented toward program development & testing.

* Advantages of load-and-go assemblers:

- It avoids the overhead of writing the object program out & reading it back in.
- Since, the object program is directly written into memory rather than some secondary storage, it becomes easier to handle forward references.
- If an instruction operand is a symbol that has not yet been defined, then the symbol is entered into the symbol table.
- This entry is flagged to indicate that the symbol is undefined.

- When the definition for a symbol is encountered, the forward reference list for that symbol is scanned and the proper address is inserted into any instructions previously generated.

(ii) Other :

- These one-pass assemblers produce object programs.
- Forward-references are entered into lists as before.
- However, when the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification.
- They will be written out as a part of Text record in the object program already.
- But later the assembler needs to again add a text record with the correct operand address.
- When the program is loaded, this address will be inserted into the instruction by the action of the loader.

* Multi-Pass Assemblers :

- It scans the ALP multiple times as required to convert into the machine code.
- One of the possible reason for this could be the symbol-definition process.

E.g

```
ALPHA   EQU  BETA
BETA    EQU  GAMMA
GAMMA   EQU  DELTA
DELTA   EQU  1
```

- For such cases, the multi-pass assembler can be a solution that can make as many passes as needed to process the definitions of symbols.
- However, it is not necessary for ~~alpha~~ an assembler to make multiple passes over the entire program.
- The portions of the program that Involve forward references in symbol definition are saved during Pass 1.
- Additional passes through these stored definition are made as the assembly progresses. This can be followed by a normal Pass 2.

-Nidhi Singh

Chapter 04: Loaders & Linkers

* **Loaders:** It is a system software that loads object program into memory for execution.

Source Program → (assembler) → Object Program → (Loader) → CPU - Memory

- An assembler converts source program into object program.
- This object program is then loaded into memory by loader.
- The loader basically brings the object program into memory & starts its execution.

* **Types of loaders:-** 02 types of loaders:
  1). Absolute loader
  2). ~~linking loader~~ A simple Bootstrap loader

(1) Absolute loader:

- It performs only loading operation i.e. it loads the object program into memory.
- It does not perform functions such as linking & program relocation. Thus, their func^n is very simple.
- All functions are accomplished in a single pass.

- It loads the object codes into memory at the same address as assigned by assembler.

Algorithm:

- The header record is checked to verify that the correct program has been presented for loading.
- As each text record is ~~check~~ read, the object code it contains is moved to the indicated add in memory.
- When End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

```
begin
    read Header record
    verify program name and length
    read first Text record
    while record type ≠ E do
        begin
        {if obj. code is in char form, convert into internal
        representation}
        move object code to specified location in memory
        read next object program record
        end
    jump to address specified in End record
end
```

② A Simple Bootstrap Loader :

- Bootstrap loader usually resides in ROM.
- When a computer is first turned on, bootstrap loader is executed.
- It loads the first program to be run by the computer which is usually an Operating System.
- The bootstrap loader itself begins at address 0 in the memory of the machine.
- It loads the operating system starting at address 80 (hex).

```
BOOT    START   0
  :
        CLEAR   A
        LDX     #128
LOOP    JSUB    GETC
        RMO     A,S
        SHIFTL  S,4
        JSUB    GETC
        ADDR    S,A
        STCH    0,X
        TIXR    X,X
        J       LOOP
```

```
GETC    TD    INPUT
        JEQ   GETC
        RD    INPUT
        COMP  #4
        JEQ   80
        COMP  #48
        JLT   GETC
        SUB   #48
        COMP  #10
        JLT   RETURN
        SUB   #7
RETURN  RSUB
INPUT   BYTE  X'F1'
        END   LOOP
```

* Machine-Dependent Loader Features :-

There are three features :- (m/c dependent)

1). Relocation.
2). Program Linking.
3). Algorithm & data structures for a linking loader.

(1) Relocation :- The loaders that allow for program relocation are called relocating loaders or relative loaders.

- For SIC → ~~for~~ Bit Mask
- For SIC/XE → Modification Record

* SIC/XE Program

0000	COPY	START	0	
0000	FIRST	STL	RETADR	17202D
0003		LDB	#LENGTH	69202D
		BASE	LENGTH	
0006	CLOOP	+JSUB	RDREC	4B101036
000A		LDA	LENGTH	032026
000D		COMP	#0	290000
0010		JEQ	ENDFIL	332007
0013		+JSUB	WRREC	4B10105D
0017		J	CLOOP	3F2FEC
001A	ENDFIL	LDA	EOF	032010
001D		STA	BUFFER	0F2016
0020		LDA	#3	010003
0023		STA	LENGTH	0F200D
0026		+JSUB	WRREC	4B10105D
002A		J	@RETADR	3E2003
002D	EOF	BYTE	C'EOF'	454F46
0030	RETADR	RESW	1	
0033	LENGTH	RESW	1	
0036	BUFFER	RESB	4096	

→ from book (end of prog)

H^COPY^000000^001077
T^000000^1D^17202D...
T...
M^000007^05+COPY
M^000014^05+COPY
M^000027^05+COPY
E^000000

* SIC Program :

0000	COPY	START 0			
0000	FIRST	STL	RETADR	140033	1
0003	CLOOP	JSUB	RDREC	481039	1
0006		LDA	LENGTH	000036	1
0009		COMP	ZERO	280030	1
000C		JEQ	ENDFIL	300015	1
000F		JSUB	WRREC	481061	1
0012		J	CLOOP	3C0003	1
0015	ENDFIL	LDA	EOF	00002A	1
0018		STA	BUFFER	0C0039	1
001B		LDA	THREE	00002D	1
001E		STA	LENGTH	0C0036	1
0021		JSUB	WRREC	481061	1
0024		LDL	RETADR	080033	1
0027		RSUB		4C0000	0
002A	EOF	BYTE	C'EOF'	454F46	0
002D	THREE	WORD	3	000003	0
0030	ZERO	WORD	0	000000	0
0033	RETADR	RESW	1		
0036	LENGTH	RESW	1		
0039	BUFFER	RESB	4096		

H.

T^000000^1E^FFC^

T^00001E^15^E00^

E^000000

→ 07×03=21 → 15 (hex)

* Program Linking :-

- When we have different control sections, the programmer has natural inclination to think of a program as a logical entity that combines all of the related control sections.
- But from programmer's prespective these are control sections — that needs to be linked, relocated & loaded.
- So, we write modification records for such cases.

PROG A

```
0000    PROGA    START   0
                 EXTDEF  LISTA, ENDA
                 EXTREF  LISTB, ENDB, LISTC, ENDC
                 :
0020    REF1     LDA     LISTA
0023    REF2   + LDT     LISTB + 4
0027    REF3     LDX     #ENDA - LISTA
                 :
0040    LISTA    EQU     *
0054    ENDA     EQU     *
0054    REF4     WORD    ENDA - LISTA + LISTC
0057    REF5     WORD    ENDC - LISTC - 10
005A    REF6     WORD    ENDC - LISTC + LISTA - 1
005D    REF7     WORD    ENDA - LISTA - (ENB - LISTB)
0060    REF8     WORD    LISTB - LISTA
                 END     REF1
```

80, Obj. Program :-

```
H...
D^LISTA^000040^ENDA^000054
R^LISTB^ENDB^LISTC^ENDC
T...
M^000024^05^+LISTB          // For REF 2
M^000054^06^+LISTC          // FOR REF 4
M^000057^06^+ENDC           } // For REF 5
M^000057^06^-LISTC          }
M^00005A^06^+ENDC           } // For REF 6
M^00005A^06^-LISTC          }
M^00005D^06^-ENDB           } // For REF 7.
M^00005D^06^+LISTB          }
M^000060^06^+LISTB          // For REF 8.

E^000020
```

* If ProgA is relocated to 4000, then ext. table would be :-

	Symbols	Add.	length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	0051
	LISTC	4112	
	ENDC	4124	

* Pass1: Assign addresses to all external symbols.
  - scans H & D record.

Pass2: Perform the actual loading, relocation and linking.
  - scans T & M record.

Pass1:
- Assigns address to all external symbols.
- Only processes Header Record & Define Record.
- Builds an external symbol table (ESTTAB) in which control section symbol is defined.

Progadder → the beginning address in memory where the linked prog is to be loaded (provided by OS).

CSADDR → the starting address assigned to the control section currently being scanned by the loader.

// Pass1, Pass2 Algorithm from book | ppt.

* Loader Design Options:

Dynamic Linking:

- Linkage loader: Linking op. is performed before prog is loaded.
- Linking loader: Linking op. is performed at the time of loading.
- However, when the linking is performed or subroutine is loaded & linked to the rest of the program when it is first called is called dynamic linking.
- It is also known as load-on-call.
- It is often used to allow several executing programs to share one copy of a subroutine or library.
- It is also used for references to software objects in an obj. oriented System.
- Dynamic linking provides the ability to load the routines only when they are needed.
- This can result into substantial savings of time & memory space.

- It avoids the necessity of loading the entire library for each execution.

// Figure from book / ppt.