ചുമ്മാ Assembler !!

നമ്മൾ എഴുതുന്ന Programs (C, C++ etc) എല്ലാം High level language ആണ്. ഇത് Machine level language-ലേക്ക് Convert ചെയ്താലേ Computer-ന് മനസ്സിലാകൂ. {Machine level language consist of 0's and 1's. That means, ON and OFF states} ഇങ്ങനെ Convert ചെയ്യാൻ ഉപയോഗിക്കുന്ന സിസ്റ്റം പ്രോഗ്രാം ആണ് **Compiler**.

ഇനി, High level language അല്ലാതെ വേറെ ഒരു സംഭവം ഉണ്ട്; 'assembly level language'. It consist of 'mnemonics' {English words such as ADD, mov etc}. ഇങ്ങനെ assembly language-നെ Machine language-ലേക്ക് Convert ചെയ്യുന്ന സംഭവം; അതായത് സിസ്റ്റം പ്രോഗ്രാം ആണ് **Assembler**.

Assembler-ന് direction കൊടുക്കുന്ന words or statements ആണ് **Assembler directives**. നമ്മുടെ Pgm-ൽ Common ആയിട്ട് use ചെയ്യുന്ന ചില assembler directives-ന്റെ function എന്താണെന്ന് നോക്കാം.

1) **START** → ഒരു Pgm-ന്റെ പേരും starting address-ഉം specify ചെയ്യും.

2) **END** → ഒരു Pgm അവസാനിച്ചു എന്ന് കാണിക്കാൻ ഉപയോഗിക്കുന്ന Assembler directive ആണ് END

3) **RESB** → statement-ൽ പറഞ്ഞിരിക്കുന്ന അത്രയും 'bytes' of memory reserve ചെയ്തു വെക്കും.

eg) RESB 4096

It means 4096 bytes of memory are reserved

ഇനി,

BUFFER RESB 4096 എന്നു പറഞ്ഞാൽ;

4096 bytes of memory are reserved for the buffer area. {BUFFER എന്നു പറയുന്നത് ഒരു label ആണ്}

4) **RESW** → Statement-ൽ പറഞ്ഞിരിക്കുന്ന അത്രയും 'words' of memory reserve ചെയ്തു വെക്കും.

5) **BYTE** → ഒന്നിച്ചു statement-ൽ പറഞ്ഞിരിക്കുന്ന character അല്ലെങ്കിൽ Hexadecimal constant ൽ വേണ്ട അത്രയും bytes-ൽ അതിനെ occupy ചെയ്യിക്കും.

eg) BYTE C'EOF'
↳ character constant

BYTE X'F1'
↳ hexadecimal constant

6) **WORD** → statement-ൽ പറഞ്ഞിട്ടുള്ള Integer constant-നെ 1 word-ൽ occupy ചെയ്യും.

eg) THREE WORD 3

ഇതിന്റെ meaning; memory-ൽ 1word space എടുത്തിട്ട് അതിന് THREE എന്ന label ചെയ്യും. (അതായത് പേരിടും). എന്നിട്ട് അതിനുള്ളിൽ 3 store ചെയ്യും.

ഇനി നമുക്ക് Pgm-ലേക്ക് കടക്കാം. ഒറ്റ ഒരു പ്രോഗ്രാം മാത്രം ഉപയോഗിച്ചാണ് നമ്മൾ Assembler-ന്റെ എല്ലാ Features-ഉം പറയാൻ പോകുന്നത്.

label	opcode	operand	
COPY	START	1000	opcode section will contain assembler directives also.
FIRST	STL	RETADR	
CLOOP	JSUB	RDREC	
	⋮		
	JSUB	WRREC	main program
	⋮		
	LDA	THREE	
	⋮		
	RSUB		
EOF	BYTE	C'EOF'	
THREE	WORD	3	
RETADR	RESW	1	
	⋮		

```
RDREC    LDX    ZERO
         LDA    ZERO      } Subroutine 1
         .
         .
         .

WRREC    LDX    ZERO
WRLOOP   TD     OUTPUT
         .
         .      } Subroutine 2
         .
         .
END      FIRST
```

Refer Figure 2.1 in the text.

Main pgm-ൽ നമ്മൾ രണ്ടു subroutines-നെ Call ചെയ്യും.

~~JSUB~~ JSUB RDREC and JSUB WRREC.

Input device-ൽ നിന്നും ഒരു record (അതായത് data) read ചെയ്യുക; ~~ഇതാണ് RDREC-ന്റെ ജോലി~~ അതിനെ Buffer-ൽ store ചെയ്യുക; ഇതാണ് RDREC-ന്റെ ജോലി. WRREC അതുപോലെ Buffer-ൽ നിന്നും ആ Record-നെ Output device ലേക്ക് Write ചെയ്യും.

ചുരുക്കം പറഞ്ഞാൽ ഒരു Record-നെ Input device-ൽ കൂടി read ചെയ്ത്, Buffer-ൽ store ചെയ്ത് അതിനെ Output device ലേക്ക് write ചെയ്യുക എന്നതാണ് നമ്മുടെ Program. എത്ര simple!! 😊

എന്തുകൊണ്ടാണ് Object code ഉണ്ടാകുന്നത്?

Assembler-ലേക്ക് Input ആയി കൊടുക്കുന്ന pgm-നെ നമ്മൾ 'Source pgm' എന്നാണ് പറയുന്നത്. ഈ Source pgm-ൽ കൂടി Assembler രണ്ടു വട്ടം Travel ചെയ്യും. അതാണ് Pass 1 and Pass 2.

{Two pass assembler-ന്റെ കാര്യമാണ് കേട്ടോ പറയുന്നത്.}

ഈ രണ്ടു pass കഴിയുമ്പോഴേക്കും നമ്മുടെ pgm-ലെ ഓരോ statements-നും corresponding ആയിട്ടുള്ള 'object code' generate ആയിരിക്കും. അത് എങ്ങനെ എന്ന് പിന്നെ പറയാം. ഈ object codes എല്ലാം കൂടി ഒരു പ്രത്യേക രീതിയിൽ

join ചെയ്തു കഴിയുമ്പോൾ 'object program' ആയി ഇതു സംഭവം Hexadecimal ആണ്. അതിനെ machine language ലേക്ക് convert ചെയ്യാൻ യാതൊരു ബുദ്ധിമുട്ടും ഇല്ല! Just use Hexadecimal to Binary Conversion.
പിന്നെന്തിനാ നമ്മുടെ Assembler ???

എന്തിനാണ് 2 passes?

Suppose, നമ്മുടെ Source pgm-ലെ ഈ രണ്ടു statements-ും നോക്കി അതിനെ object code-ലേക്ക് convert ചെയ്താണ് Assembler പോകുന്നത് എന്ന് കരുതുക. അങ്ങനെയാണെങ്കിൽ ചില ❌ വരുകൾ Assembler ന് യാതൊരു പരിചയവും കാണില്ല.

For eg; Refer **figure 2.2**

line no:			
	COPY	START	1000
10	1000 FIRST	STL	RETADR
	⋮		
95	1033 RETADR	RESW	1

COPY & FIRST - labels ആണ്
START - assembler directive
STL - machine instruction

പക്ഷേ RETADR എന്താണെന്ന് Assembler ന് മനസ്സിലാകില്ല. Because അതെന്താണ് എന്ന് പറഞ്ഞിരിക്കുന്നത് Pgm-ന്റെ പിന്നീടുള്ള ഭാഗത്താണ് {1033 എന്ന address-ൽ}
ടം Assembler ആദ്യമേ തന്നെ object code എഴുതി പോവുകയാണെങ്കിൽ RETADR വരുമ്പോൾ Stuck ആകും!
ഈ ഒരു പ്രശ്നത്തെയാണ് നമ്മൾ **forward reference** എന്നു പറയുന്നത്.
ഇതൊഴിവാക്കാനാണ് 2 passes.
'എങ്ങനെ' എന്നു പഠിക്കുന്നതിനു മുമ്പ് നമുക്ക്

Assembler ഉപയോഗിക്കുന്ന രണ്ട് Data Structure-നെ കുറിച്ച് പഠിക്കാം.

1) Operation Code Table (OPTAB)
2) Symbol Table (SYMTAB)

OPTAB - ൽ നെമോണിക് opcode-ന്റെ (LDA, STL, JSUB etc) machine language equivalents ആണ്ടിലാകും.

eg) LDA → 00 { Refr Appendix A at the
STL → 14 back of text }

ഒരു Assembler ഉണ്ടാക്കുമ്പോൾ തന്നെ അതിന്റെ OPTAB - ൽ ഈ informations ഒക്കെ define ചെയ്തിരിക്കും. ഇതു കൂടാതെ OPTAB - ൽ ചിലപ്പോൾ Instruction formats - ഉം അതിന്റെ length - ഉം ഒക്കെ ചേർത്തിട്ടുണ്ടാകാം.

SYMTAB - ൽ ഓരോ label - ഉം അതിന്റെ address - ഉം ആണ് ഉള്ളത്. Error conditions കാണിക്കാനുള്ള flags - ഉം ഉണ്ടായിരിക്കും.

അതായത്, LDA - യ്ക്ക് പകരം Pgm - ൽ LAD എന്നാണ് കൊടുത്തതെങ്കിൽ error കാണിക്കണ്ടേ? അതിനു വേണ്ടിയുള്ള set-up ഒക്കെ SYMTAB - ൽ കാണും.

ഇനി നമുക്ക് തിരിച്ച് Passes - ലേക്ക് പോകാം.

ആദ്യത്തെ Pass - ൽ ~~ആദ്യ~~ ~~sta~~ source pgm - ലെ എല്ലാ statements നും address assign ചെയ്യും. (incremented by 3). അതായത് ~~ആദ്യത്തെ~~ ~~statement~~ starting address 1000 ആണെങ്കിൽ 1000, 1003, 1006, 1009, 100C, 100F, 1010, ... എന്നായിരിക്കും ബാക്കിയുള്ള addresses. അതിന്റെ Reason എന്താണെന്ന് വെച്ചാൽ; നമ്മൾ പഠിക്കുന്നത് SIC, SIC/XE machine - ന്റെ assembler ആണ്. ഈ machine - ന്റെ wordlength 3 bytes ആണ്. So incremented by 3.

അപ്പോൾ നമ്മൾ Address ~~ആ~~ Assign ചെയ്തു കഴിഞ്ഞു. ഇനി ഓരോ symbol - ന്റെയും address symbol table - ൽ store ചെയ്യണം.

ഒരോ line-ഉം scan ചെയ്ത് labels-ന്റെ address SYMTAB-ൽ store ചെയ്ത് source program ന്റെ അവസാനം എത്തുമ്പോൾ വീണ്ടും starting-ൽ പോവുക. Because its the time for Pass 2!!

Pass 2-ൽ ആണ് ശരിക്കുമുള്ള object code generation നടക്കുന്നത്. Pass 1 കഴിയുമ്പോൾ നമ്മുടെ source pgm നെ ഒരു intermediate file-ലേക്ക് write ചെയ്തിട്ടുണ്ടാവും. ഇതാണ് Pass 2-ൽ input ആയിട്ട് കൊടുക്കുന്നത്.

Let us consider the statement

STL RETADR.

STL-ന്റെ machine equivalent OPTAB നിന്നും കിട്ടും. It is 14.

RETADR-ന്റെ address SYMTAB-ൽ നിന്നും കിട്ടും. It is 1033.

∴ Object code for the statement STL RETADR is 141033

ഇങ്ങനെ എല്ലാ statements-ന്റെയും object code എഴുതാൻ പറ്റും. {Refer fig 2.2}

ഈ object codes എല്ലാം കൂടി ചേർത്താണ് object program ഉണ്ടാക്കുന്നത്. Refer figure 2.3

ഒരു സാധാരണ object program format-ൽ മൂന്ന് തരം records ഉണ്ടാവും. Header record, Text record & End record.

Header record is denoted by H. ഇതിൽ Pgm-ന്റെ പേര്, starting address ഉം length ഉം ഉണ്ടായിരിക്കും.

eg) H^COPY^001000^00107A

Header → Pgm name ← Pgm name

→ length of pgm

→ starting address

Text record – denoted by T. ഇതിലായിരിക്കും object code. കൂടാതെ ഒരു particular text record-ന്റെ starting address for object, പിന്നീട് ആ text record ന്റെ length, ഇതും കഴിഞ്ഞ്

ചെയ്തുള്ള object codes. (Refer fig 2.3)
End record marks the end of an object program. It is denoted by E; and contains starting address of the program.

ഉദാ) E^001000

{ ^ symbol seperation കാണിക്കാൻ വേണ്ടി ഉപയോഗിക്കുന്നതാണ്. അത് object pgm-ന്റെ part ആണെന്ന് വിചാരിക്കരുതേ! }

ഇനി വേറൊരു കാര്യം; LOCCTR എന്നൊരു variable നമ്മൾ പല സ്ഥലങ്ങളിലും കാണും. അതാണ് Location Counter. ഇതിന്റെ ഉള്ളിൽ starting address store ചെയ്യുക and increment by 3 while executing each statement. അതായത്, നമ്മൾ execute ചെയ്യുന്ന ഓരോ statement-ന്റെ address വേണമെങ്കിൽ LOCCTR-ലെ value എടുത്താൽ മതി. OK? അതു മനസ്സിലായില്ലേ.. ഇനി Pass 1 and Pass 2 algorithms നോക്കാം-

Algorithm for pass 1 of assembler

(1) Source pgm-ന്റെ ആദ്യത്തെ line വായിക്കുക.
(2) അതിന്റെ opcode 'START' ആണെങ്കിൽ operand field-ൽ നിന്നിരിക്കുന്ന value starting address ആയിട്ട് save ചെയ്യുക. ഈ value തന്നെ LOCCTR നും കൊടുക്കുക. ഇനി opcode 'START' അല്ലെങ്കിൽ LOCCTR = 0
(3) അടുത്ത line വായിക്കുക.
(4) Label ഉണ്ടെങ്കിൽ SYMTAB check ചെയ്യുക. { checking whether label is already present } Label already ഉണ്ടെങ്കിൽ 'duplicate symbol' എന്ന് error flag set ചെയ്യുക; ഇല്ലെങ്കിൽ label-ഉം അതിന്റെ address-ഉം symbol table-ൽ insert ചെയ്യുക.
(5) ഇനി opcode-ന്റെ കാര്യം നോക്കാം. statement-ൽ opcode ഉണ്ടെങ്കിൽ അതിന്റെ OPTAB നോക്കും. If it is valid then LOCCTR is incremented by 3. എന്നിട്ട് നോക്കും opcode എന്താണെന്ന്.

(6) OPCODE = WORD ആണെങ്കിൽ add 3 to LOCCTR. Because ആ statement നോക്കണ്ട കാര്യമില്ല.

(7) OPCODE = RESW ആണെങ്കിൽ statement-ൽ പറഞ്ഞിരിക്കുന്ന അത്രയും words reserve ആകണ്ടേ? ie 3 x #[operand] is added to LOCCTR. 3 എന്നത് ഒരു word-ന്റെ length ആണ് (ie 3 bytes) ie 2 words reserve ചെയ്യണമെങ്കിൽ 3x2 bytes reserve ചെയ്യണം. അതായത് LOCCTR 6 ആഡ് increment ചെയ്യണം.

(8) OPCODE = RESB ആണെങ്കിൽ പറഞ്ഞിരിക്കുന്ന അത്രയും byte. LOCCTR-ന്റെ കൂടെ add ചെയ്യുക.

(9) OPCODE = BYTE ആണെങ്കിൽ constant-ന്റെ length കണ്ടുപിടിക്കുക. എന്നിട്ട് add to LOCCTR.

(10) ഇതൊന്നുമല്ലെങ്കിൽ error!

(11) അടുത്ത line വായിക്കുക; same procedure repeat until end of pgm.

(12) ഓരോ statement-ഉം നോക്കി കഴിഞ്ഞ് Intermediate file-ലേക്ക് write ചെയ്യണം.

അങ്ങനെ Pass 1 കഴിഞ്ഞപ്പോഴേക്കും നമ്മുടെ source pgm-ൽ ഉള്ള എല്ലാ symbols ഉം അതിന്റെ address-ഉം Symbol table-ൽ store ആയി. ഇനി നേരത്തെ നമ്മൾ പറഞ്ഞതുപോലെ രണ്ടാമത്തെ Pass-ൽ source pgm-ലെ ഓരോ statement ഉം എടുത്ത് ~~correct~~ അതിന്റെ object code generate ചെയ്യാൻ പറ്റും. {OPTAB നോക്കി mnemonic-ന്റെ machine equivalent എടുക്കുക; SYMTAB നോക്കി labels-ന്റെ address എടുക്കുക}. ഇനി object codes എല്ലാം കൂടി ചേർത്ത് Header, Text, End record format-ൽ Object program എഴുതുക. ഇതാണ് Pass 2. ഇനി നമുക്ക് Pass-2 algorithm detail ആയി നോക്കാം.

Algorithm for pass 2 of assembler:

(1) Intermediate file-ൽ നിന്നും ആദ്യത്തെ line വായിക്കുക.

(2) അതിന്റെ OPCODE 'START' ആണെങ്കിൽ

നമുക്ക് object program എഴുതി തുടങ്ങാം. So read next input line from intermediate file.

(3) ~~Header~~ Object program-ൽ Header record എഴുതുക. It should start with 'H' and must contain the name of the program, starting address and its length.

(4) ഇനി നമുക്ക് ആദ്യത്തെ Text record എഴുതാം. object code ന്റെ starting address വെച്ച് initialize ചെയ്യുക. നമ്മുടെ Pgm-ൽ അത് 1000 ആണ് (Refer Fig 2.2). ഇനി ~~object~~ അതിന് ശേഷം Text record-ന്റെ length-ഉം പിന്നീട് object codes-ഉം എഴുതണം. {ഒരു text record ന്റെ maximum length 60 characters ആണ്. ഇത് കഴിഞ്ഞും object code എഴുതാനുണ്ടെങ്കിൽ അടുത്ത Text record initialize ചെയ്യണം.

For eg; Fig 2.3-ലെ 1st Text record ആണ് $T\wedge 001000\wedge 1E\wedge 141033\wedge \ldots\ldots$ ഇതിൽ 001000 എന്നത് object code-ന്റെ starting address-ഉം 1E എന്നത് ഈ particular text record-ന്റെ length-ഉം ആണ്. അത് കഴിഞ്ഞു വരുന്നതാണ് object codes (ആദ്യത്തെ object code 141033). ഈ object codes എണ്ണി നോക്കിയാൽ 60 characters ആണെന്ന് മനസ്സിലാകും; which is the maximum length of the text record. അത് കഴിഞ്ഞുള്ള object codes അടുത്ത Text record initialize ചെയ്താണ് കൊടുത്തിട്ടുള്ളത്. ie,

$T\wedge 00101E\wedge 15\wedge 0C1036\wedge \ldots\ldots$

starting address ↵ ↳ length

ഇവിടെ 15 അയത് എന്താണെന്ന് വെച്ചാൽ {1039 − 101E = 15 ; അല്ലാതെ decimal 15 അല്ല}. ഈ 15 കഴിഞ്ഞ പ്പോഴും നമ്മുടെ main pgm തീർന്നു. അതുകൊണ്ടാണ് object pgm-ൽ ബാക്കി space വന്നത്. ഇനിയുള്ള subroutines അടുത്ത Text record-ൽ തുടങ്ങും. ie,

$T\wedge 002039\wedge 1E\wedge 041030\wedge \ldots\ldots$

അങ്ങനെ full object codes-ഉം Text record-ൽ എഴുതും. } ഇനി നമുക്ക് Algorithm വെച്ച് പറയാം.

ആദ്യമേ object code കണ്ടുപിടിക്കണം; എന്നിട്ട് അത് Text record-ൽ എഴുതണം.

(a) S. 1st statement എടുക്കുക. അതിന്റെ opcode OPTAB-ൽ ഉണ്ടോ എന്നു നോക്കുക. ഉണ്ടെങ്കിൽ opcode ~~is~~ valid ആണ്. ~~ഇനി operand-ലെ symbol SYMTAB-ൽ ഉണ്ടോ എന്നു നോക്കുക. ഉണ്ടെങ്കിൽ symbol~~ അതിന്റെ machine equivalent എടുക്കുക. ~~ഇനി~~

(b) ഇനി operand-ലെ symbol SYMTAB-ൽ ഉണ്ടോ എന്നു നോക്കുക. ഉണ്ടെങ്കിൽ അതിന്റെ address SYMTAB-ൽ നിന്നും എടുക്കുക; ഇല്ലെങ്കിൽ undefined symbol എന്ന് error കാണിക്കുക.

(c) ഇത് രണ്ടും കൂടി join ചെയ്യുക. {example നോക്കിയിട്ടുണ്ട്; STL RETADR}
ie, assemble the object code instruction

(d) ഈ object code നമ്മുടെ current Text Record-ൽ fit ചെയ്യുമോ എന്ന് നോക്കുക. ചെയ്യുമെങ്കിൽ അത് അവിടെ എഴുതുക. ഇല്ലെങ്കിൽ പുതിയ Text record initialize ചെയ്ത് അതിനുശേഷം എഴുതുക.

ഇങ്ങനെ ഓരോ statement-ന്റെയും object code text record-ൽ എഴുതുക.

(5) അവസാനത്തെ Text record-ഉം എഴുതിയതിന് ശേഷം End record എഴുതുക.

അങ്ങനെ ~~വിജയകരമായി~~ നമ്മൾ source pgm-നെ object pgm-ലേക്ക് വിജയകരമായി convert ചെയ്തു കഴിഞ്ഞു.

ഇതാണ് Assembler !!

ഇതുപോലെ നമ്മുടെ pgm SIC assembler language-ൽ തന്നിരുന്നു. ഇനി നമുക്ക് SIC/XE-ൽ വരുമ്പോൾ എന്തു മാറ്റം വരും എന്ന് നോക്കാം. Text book തുറക്കുക, and compare figure 2.5 and figure 2.1. രണ്ടും same pgm തന്നെ. പക്ഷേ രണ്ട് assembly language-ൽ എഴുതിയിരിക്കുന്നതാണ് മാത്രം. figure 2.1-ൽ പറഞ്ഞിരിക്കുന്നത് ഒരു SIC machine-ന് മനസ്സിലാകുന്ന pgm തന്നെ. figure 2.5-ൽ പറഞ്ഞിരിക്കുന്നതോ SIC/XE machine-ന് മനസ്സിലാകുന്നതും! ഇങ്ങനെ ഓരോ machines-നും മനസ്സിലാകുന്ന assembly languages ഉണ്ട് (1st module-ൽ നമ്മൾ 7 machine architecture പഠിച്ചത് ഓർക്കുന്നില്ലേ? SIC, SIC/XE, VAX, CRAY etc). ഇതിൽ SIC & SIC/XE-യുടെ assembly language മാത്രം നമുക്ക് ശ്രദ്ധിച്ചാൽ മതി. അതെ പഠിക്കാനുള്ളൂ :))

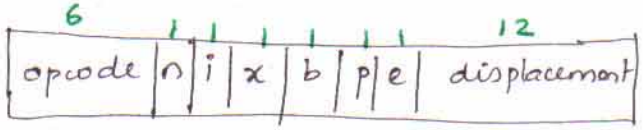
→ SIC/XE ~~ൽ~~ pgm-ൽ base register ഉപയോഗിച്ചിട്ടുണ്ട് (line 12 & 13 of fig 2.5);

→ immediate addressing ഉപയോഗിച്ചിട്ടുണ്ട് (using #) eg) line 133

→ ഇവിടെ നാല് Instruction formats ഉണ്ട്; Format 1, 2, 3 and 4. (1st module നോക്കിയാൽ മനസ്സിലാകും.) ഇതിൽ main ആയിട്ടും Format 3 ~~and~~ or Format 4-ൽ ആയിരിക്കും നമ്മുടെ pgm ലെ ഒരോ instructions-ഉം.

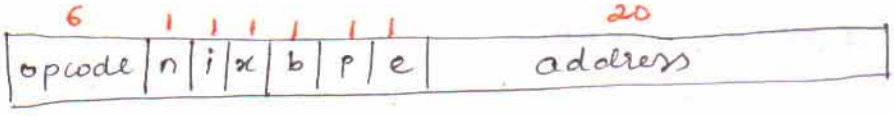
Format 3 :- relative addressing

So total 24 bits = 3 bytes



Format 4 :- extended addressing

So total 32 bits = 4 bytes



n, i, x, b, p, e ~~ൽ~~ ഇത് flag registers ആണ് (of SIC/XE machine). ഇതിൽ ഓരോന്നിനും ഓരോ purpose ഉണ്ട്.

→ e bit 1 ആണെങ്കിൽ Format 4; 1 അല്ലെങ്കിൽ format 3

→ p bit 1 ആണെങ്കിൽ നമ്മുടെ instruction-ൽ ഉപയോഗിച്ചിരിക്കുന്ന addressing mode PC relative ആയിരിക്കും.

→ b bit 1 ആണെങ്കിൽ addressing mode is base relative

{At a time ഒന്നുകിൽ p=1; അല്ലെങ്കിൽ b=1. ഇത് രണ്ടും കൂടി ഒരുമിച്ച് 1 ആകില്ല.}

→ x bit 1 ആണെങ്കിൽ Index addressing mode-ഉം ഉപയോഗിച്ചിട്ടുണ്ടാവും.

{ഈ mode ഉപയോഗിച്ചെങ്കിൽ നമ്മുടെ instruction-ൽ X ഉണ്ടാവും. For eg: (in line no: 160)

STCH BUFFER, X

→ ~~i=0~~ i=1 and n=0 means immediate addressing

{instruction-ൽ # symbol ഉണ്ടായിരിക്കും.}

For eg (in line no: 55)

LDA #3

→ i=0 and n=1 means indirect addressing

{instruction-ൽ @ symbol ഉണ്ടായിരിക്കും.}

For eg (in line no: 70)

J @ RETADR

→ ഇനി immediate-ഉം indirect-ഉം ഉപയോഗിച്ചിട്ടില്ലെങ്കിൽ ഈ രണ്ടു bit-ഉം 1 ആയിട്ട് set ചെയ്യണം.

ie i=1 and n=1

നിൽക്കുന്ന instruction Format 3 ആണോ അതോ Format 4 ആണോ എന്ന് എങ്ങനെ അറിയുക? mnemonic-ന്റെ കൂടെ + symbol ഉണ്ടെങ്കിൽ അത് Format 4 ആയിരിക്കും!

eg) Line no 35

+JSUB WRREC

SIC/XE program-ന്റെ object code എഴുതുന്നത് എങ്ങനെ?

SIC-ടെ object code ഉ എഴുതുന്നതു പോലെ അത്ര Simple അല്ല SIC/XE-ടെ എഴുതാൻ. അവിടെ mnemonic-ന്റെ machine equivalent-ഉം operand address-ഉം കൂടി join ചെയ്ത് എഴുതിയാൽ ജോലി തീർന്നു! പക്ഷേ ഇവിടെ അങ്ങനെയല്ല. കുറച്ചു കൂടി പണിയുണ്ട്. B'coz അവിടെ ഒരു Instruction format-ഉം രണ്ട് addressing mode-ഉം (Direct and index) മാത്രമേ ഉണ്ടായിരുന്നുള്ളൂ. അതേ സമയം ഇവിടെ 'നാല്' instruction format-ഉം 6 addressing modes-ഉം ഉണ്ട്.

(Direct, Index, Base relative, PC relative, immediate, indirect)

So, ഒരു instruction ന എഴുതാൻ ആദ്യം അത് ഏതു Format-ൽ ആണെന്ന് നോക്കണം; പിന്നെ ഏതു addressing mode ആണെന്ന് നോക്കണം. ഓരോ Formats-നും ഓരോ addressing modes-നും object code എഴുതുന്നത് പല രീതിയിൽ ആണ്. ☺ പേടിക്കണ്ട; സംഭവം Simple ആണ്. ഒ മനസ്സിലാക്കണം എന്നേയുള്ളൂ.

അപ്പൊ തുടങ്ങാം? OK. Take Figure 2.6

ആദ്യമേ നമുക്ക് ഒരു Register to Register instruction എടുക്കാം. അത് Format 2 ആണ്.

8	4	4
opcode	register 1	register 2

So total 16 bits = 2 bytes

For eg; line no: 150

COMPR A, S

ആദ്യം COMPR-ന്റെ machine equivalent എഴുതണം { It is 'A0'. { Text-ന്റെ ~~[illegible]~~ പുറകിൽ ഉണ്ട്. Appendix A-ൽ } Assembler അത് OPTAB-ൽ നിന്നും എടുക്കേണ്ടതാണ്. ഇനി registers; അതിന്റെ equivalent SYMTAB-ൽ നിന്നും എടുക്കും.

A - 0	B - 3	T - 5
X - 1	S - 4	F - 6
L - 2		

A	- 0
X	- 1
L	- 2
B	- 3
S	- 4
T	- 5
F	- 6

ഈ values SIC/XE assembler-ന്റെ SYMTAB-ൽ നേരത്തെ തന്നെ define ചെയ്തിരിക്കും.

ex: COMPR A,S -ന്റെ object code is: A0 0 4

COMPR → A, S

Check line no: 150. Object code A004 അല്ലേ?

ഇനി നമുക്ക് Register-ഉം Memory-ഉം കൂടി വരുന്ന Instructions നോക്കാം.

ie Register to Memory Instructions

ഇവിടെ നേരത്തെ Format 3 ആയിരുന്ന Format 4-ൽ ആയിരിക്കും instructions.

അടുത്തതായി നമുക്ക് Format 3-യുടെ കാര്യം നോക്കാം.

Addressing modes PC relative or base relative ആകാം. Normally PC relative ആണ് ഉപയോഗിക്കുന്നത്.

രണ്ടും തമ്മിലുള്ള വ്യത്യാസം എന്താണെന്ന് വെച്ചാൽ;

1) PC relative-ന്റെ displacement കണ്ടുപിടിക്കാനുള്ള formula

TA - Target Address

$$disp = TA - \text{content of } PC$$

ഇങ്ങനെ കിട്ടുന്ന displacement value -2048 മുതൽ +2047 വരെ ഇടയ്ക്ക് ആയിരിക്കും.

ie: $\boxed{-2048 \leq disp \leq 2047}$

അതേ സമയം base relative-ൽ PC-യ്ക്ക് പകരം base register B ആണ് ഉപയോഗിക്കുന്നത്.

$$displacement = TA - \text{content of } B$$

ഇവിടെ disp value 0-യും 4095-നും ഇടയ്ക്ക് ആയിരിക്കും.

value of base register is given by user

$$\boxed{0 \leq disp \leq 4095}$$

ഇങ്ങനെ opcode-ഉം flag bits-ഉം displacement-ഉം കൂടി assemble ചെയ്ത് കിട്ടുന്നതാണ് object code ആയി!

ഒരു example നോക്കാം. Consider line no: 160

```
STCH   BUFFER, X
```

ഇനി നമുക്ക് STCH എന്ന opcode-ന്റെ machine equivalent എന്താണെന്ന് നോക്കാം. It is 54. ഇനി flag bits-ഉം displacement-ഉം കൂടി എഴുതിയാൽ object code റെഡിയായി! flag bits എഴുതുന്നതിനു മുമ്പ് displacement കണ്ടുപിടിക്കുന്നത് എങ്ങനെയാണെന്ന് പഠിക്കാം.

Normally, PC relative addressing mode ആണ് ഉപയോഗിക്കുന്നത് എന്ന് നേരത്തെ പറഞ്ഞിരുന്നല്ലോ. നമുക്ക് ഇപ്പോൾ ഇവിടെ base relative ആയിട്ട് എടുക്കാം. ടം

$$disp = TA - (B)$$

ഇതാണ് നമ്മുടെ formula. Target address എന്ന് പറഞ്ഞാൽ നമ്മുടെ symbol (ie BUFFER) define ചെയ്തിരിക്കുന്ന location. Clearly, it is 0036 from line no: 105. ഇനി, content of base register കണ്ടുപിടിക്കണം. അതിനുവേണ്ടി നമ്മുടെ p. source pgm-ൽ BASE എന്ന Assembler directive എവിടെയാണ് എന്ന് നോക്കുക. It is in line no: 13.

```
BASE   LENGTH
```

ഇനി LENGTH എവിടെയാണ് define ചെയ്തിരിക്കുന്നത് എന്ന് നോക്കുക. I mean in which location. Clearly, it is in line no: 100.

```
0033   LENGTH   RESW   1
```

ഈ location, അതായത് 0033 ആണ് content of base register. ടം ഇനി displacement കണ്ടുപിടിക്കാം.

$$displacement = 0036 - 0033$$
$$= 0003$$

ഈ displacement 12 bit ആയിട്ട് തന്നെ വേണം. object code ൽ represent ചെയ്യാൻ. ie,

0000 0000 0011

ഇനി flag-bits-ന്റെ കാര്യം നോക്കാം.
Immediate & indirect addressing ഉപയോഗിച്ചിട്ടില്ല.

So n=1 ; i=1
Index addressing ഉണ്ട്. So x=1
Base relative ആണ് ഉപയോഗിച്ചത്. So b=1
PC relative ഉപയോഗിച്ചിട്ടില്ല. So p=0
Format 3 ആയത് കൊണ്ട് e=0.

Object code എഴുതാനുള്ള എല്ലാ സാമഗ്രികളും ആയില്ലേ?? STCH ന്റെ machine equivalent കിട്ടി; flag bits എല്ലാം ~~എങ്ങനെയാണ~~ - ൽ ഏതൊക്കെ 1 ആയി set ചെയ്യണം എന്ന് മനസ്സിലായി; displacement കണ്ടുപിടിച്ചു. ഇനി object code എഴുതാൻ എന്താ പ്രയാസം ☺
എങ്ങനെയാ assemble ചെയ്യുന്നത് എന്ന് നോക്കിക്കോളൂ.

1)

n	i	x	b	p	e		displacement
0101	0100	1	1	1	0	0	0000 0000 0011
5	4						

2) opcode-ന്റെ last രണ്ട് bit-ന്റെ സ്ഥാനത്ത് n & i കയറിയിരിക്കും.

n	i	x	b	p	e		displacement
0101	01(11)	1	1	0	0	0000 0000 0011	
5							

ഇപ്പോ correct 24 bits ആയില്ലേ.
(ie 3 bytes)

ഇനി നമുക്ക് ഈ കിട്ടിയ സംഭവം എന്താണെന്ന് നോക്കാം.

0101	0111	1100	0000	0000	0011
5	7	C	0	0	3

ie 57C003

ഇതാണ് STCH BUFFER,X എന്ന Instruction ന്റെ object code. {See line no: 160}
മനസ്സിലായല്ലോ?? 😊

ഇനി നമുക്ക് Format 4 - ൽ ഉള്ള Instruction-ന്റെ Object code എങ്ങനെയാ എഴുതുന്നത് എന്ന് നോക്കാം. ഞാൻ നേരത്തെ പറഞ്ഞിരുന്നു; Format 4 ആണെങ്കിൽ instruction-ൽ mnemonic-ന്റെ കൂടെ '+' symbol കാണുമെന്ന്. അത് മനസ്സിലുണ്ടാകുമല്ലോ.
So, let us consider an example (line no: 15)

```
+JSUB   RDREC
```

ഇത് Format 3 - യിലാണ് എഴുതിയതാണ്. Because ഇവിടെ displacement തരാം കണ്ടുപിടിച്ച് ബുദ്ധിമുട്ടുണ്ട്! symbol-ന്റെ (ഇവിടെ symbol is RDREC) address direct ആയിട്ട് അങ്ങ് എഴുതിക്കാൻ മതി. പക്ഷേ address 20 bit-ൽ തന്നെ എഴുതണം.
അതായത്, ഇവിടെ RDREC-ന്റെ Address ആണ് 1036 (line no: 125)
That is

```
          1    0    3    6
0000  0001 0000 0011 0110
      \_____20 bits_____/
```

ഇങ്ങനെ വേണം address എഴുതാൻ. ബാക്കി പഴയതു പോലെ തന്നെ. JSUB-ന്റെ machine equivalent എഴുതുക. {It is 48} Flags set ചെയ്യുക.

```
n  i  x  b  p  e
1  1  0  0  0  1
\__/           \
neither          ↳ extended format
immediate          (Format 4)
nor is direct
```

അതു കഴിഞ്ഞ് address-ഉം എഴുതുക.
So object code is

```
          n i x b p e
0100  1000  110001  0000 0001 0000 0011 0110
\__/  \__/          \_____disp address_____/
 4     8
```

opcode (അതായത് 48)-ന്റെ last 2 bit കളയുക.

```
          ni  x b p e
0100  1011  0001  0000 0001 0000 0011 0110
\__/                \_____address_____/
 4
```

ഇത് എന്താണെന്ന് നോക്കാം.

0100 1011 0001 0000 0001 0000 0011 0110

4 B 1 0 1 0 3 6

ie 4B101036. This is the object code of the instruction +JSUB RDREC.

eg 2) LDA #3 (line no 55)

- Machine equivalent of LDA is 00
- Immediate addressing is used. So i=1 & n=0 അതുകൊണ്ട് flags ന്റെടുത്ത് '0'. {Format 3. So e=0}
- #3 is 0011 ~~in hexadecimal~~.
  അത് 12 bit ന്റെടുത്ത് represent ചെയ്യണം. (Since disp is 12 bit in format 3)

n i x b p e

0000 0000 0100 0000 0000 0011

00 3

Opcode is

n i x b p e

0000 0001 0000 0000 0000 0011

0 1 0 0 0 3

= 010003

eg 3) +LDT #4096 (line no: 133)

- + symbol ഉള്ളതുകൊണ്ട് Format 4. ie e=1
- Machine ~~ado~~ equivalent of LDT is 74
  ie $(0111\ 0100)_H$
- Immediate addressing is used. So i=1 & n=0
- Hexadecimal value of 4096 is $(1000)_H$
  {Decimal -ൽ Hexadecimal ന്റെടുത്ത് convert ചെയ്യണമെങ്കിൽ ÷ by 16}

16 | 4096 — 0
16 | 256 — 0
16 | 16 — 0
1

n i x b p e

0111 0100 0001 0000 0001 0000 0000 0000

7 4 ~~address~~ ie 1000

Object code is

0111 0101 0001 0000 0001 0000 0000 0000

7 5 1 0 1 0 0 0

= 75101000

eg 4) `J @RETADR` (line no: 70)

- + symbol ഇല്ലാത്തതു കൊണ്ട് Format 3
- J - ഉടെ machine equivalent 3C
- Indirect addressing ആണ്. So n = 1 & i = 0
- ഇനി displacement കണ്ടുപിടിക്കണം. Assume PC relative

TA = 0030 (from line no: 95)

[PC] = 002D (just above 0030)

∴ displacement = TA − [PC]

= 0030 − 002D

= 0003

0011 1100 10 0000 0000 0011

3 C n i x b p e disp

Object code is

0011 1110 0010 0000 0000 0011

3 E 2 0 0 3

= 3E2003

പാഠം Assembler (ഭാഗം-2)

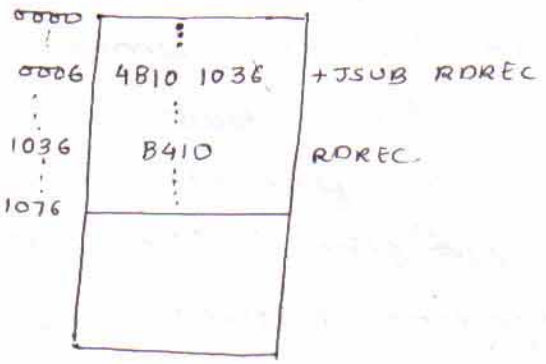
Assembler എന്താണെന്നും, Assembler എന്ന system program കൊണ്ടുള്ള ഉപയോഗം എന്താണെന്നും നമ്മൾ Part-1-ൽ പഠിച്ചു. അറിയാമല്ലോ അല്ലേ?? ☺ OK. ഇനി നമുക്ക് Assembler-ന്റെ 'features' എന്തൊക്കെയാണെന്ന് പഠിക്കാം.

Assembler features നമ്മൾ രണ്ടായിട്ട് classify ചെയ്താണ് പഠിക്കാൻ പോകുന്നത്. Machine dependent features-ഉം Machine Independent feature-ഉം! See, ഓരോ Machine-നും ഓരോ type assembly language ആണെന്ന് നമ്മൾ ~~കണ്ടതാണ്~~ പഠിച്ചു. (അതായത് Assembler) ആണെന്ന് നമ്മൾ കണ്ടതാണ് പഠിച്ചു. For eg, SIC Machine-ൽ use ചെയ്യുന്ന assembly language അല്ല SIC/XE-ൽ! Extra കുറേ സംഗതികളൊക്കെ വന്നില്ലേ?? (Eg Instruction formats and Addressing modes; അതെന്താ?? ☹) '+' '@' '#' ഈ symbols ഒക്കെ കണ്ടാൽ SIC machine അല്ല! B'coz ഇതൊക്കെ 'SIC/XE' machine-ന്റെ സ്വന്തം symbols ആണ്. ഈ SIC/XE assembler-നു ഈ symbol-ന്റെ meaning മനസ്സിലാവും. Similarly VAX machine-ന്റെ assembly language കണ്ടാൽ SIC/XE-ക്ക് ഒന്നും മനസ്സിലാവില്ല. B'coz അവിടെ use ചെയ്തിരിക്കുന്ന Instruction formats and Addressing modes different ആണ്. ചുരുക്കം പറഞ്ഞാൽ ഒരു Assembler, Instruction formats-ഉം Addressing modes-ഉം ഒക്കെ തിരിച്ചറിയുന്നത് ആ Assembler ഏതു machine-ന്റെ ആണ് എന്നതിനെ ആശ്രയിച്ചിരിക്കും. (VAX Assembler-ന് SIC/XE-ടെ Instruction formats മനസ്സിലാവില്ല) That means, **Instruction formats & Addressing Modes** are the **Machine Dependent** features of Assembler. ഇപ്പോ കാര്യം പിടികിട്ടിയോ?? OK. ഇനി വേറെ ഒരു Machine Dependent Assembler feature

കൂടി ഉണ്ട്. **Program Relocation**!! അതെന്താണെന്ന് വെച്ചാൽ; See, നമ്മുടെ object program നമ്മൾ memory-ൽ load ചെയ്തിട്ടാണ് execute ചെയ്യുന്നത്. {This loading process is done by 'LOADER'. അത് പിന്നെ പഠിക്കാം! } Memory-ൽ എവിടെ load ചെയ്യണം എന്ന് Object program-ലെ Header Record-ൽ പറഞ്ഞിട്ടുണ്ട്; as starting address of the program. right? പക്ഷേ, പറഞ്ഞിരിക്കുന്ന Address-ൽ തന്നെ load ചെയ്യണം എന്ന് നിർബന്ധമില്ല! B'coz അവിടെ; ഈ അഡ്രസ് 1000 തന്നെ Memory location-ൽ Already വേറെ ഏതെങ്കിലും Pgm ഉണ്ടെങ്കിൽ എന്തു ചെയ്യും?? ഒന്നും ചെയ്യാനില്ല; വേറെ എവിടെ space ഉണ്ടോ അവിടെ load ചെയ്യും! ☺

That is, it is desirable to load a program into the memory wherever there is room for it. And in such a situation, the actual starting address of the program is not known until the load time. (Pgm-ൽ starting address 1000 ~~ആയിരിക്കും~~. അതിരുന്നാ പക്ഷേ load ചെയ്യാൻ പറ്റിയത് 4000 എന്ന memory location-ൽ ആണെങ്കിൽ starting address 2000 ആയിട്ടല്ലേ? അതാ പറഞ്ഞത്; ഇങ്ങനെയൊരു situation-ൽ load ചെയ്തുകഴിയുന്നതുവരെ Pgm-ന്റെ starting address അറിയാൻ പറ്റില്ല!)

പക്ഷേ ഇവിടെ ഒരു പ്രശ്നം ഉണ്ട്. Consider fig. 2.7. Fig. 2.6-ലെ pgm memory-ൽ ~~load~~ ~~ചെ~~ ചില locations-ൽ load ചെയ്ത diagram ആണ് ഇത്.

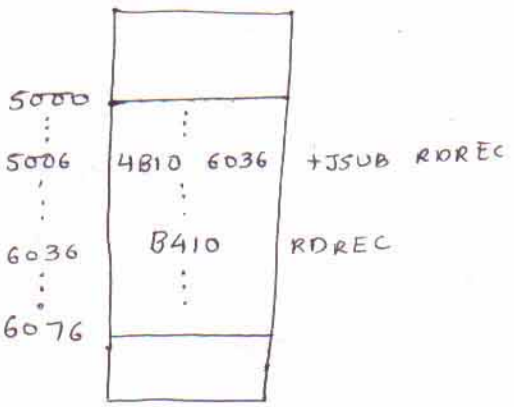


(a)

ഇവിടെ 0000 എന്ന memory location-ൽ തന്നെ ആണ് Pgm load ചെയ്തത്. So No problem!
4B10 1036 എന്ന് കാണുമ്പോൾ 1036-ലേക്ക് 'JUMP' ചെയ്തോളും. അതായത് RDREC-ലേക്ക്!

{ object pgm ആണ് memory-ൽ load ചെയ്യുന്നത്. So അതുകൊണ്ടാണ് +JSUB RDREC എന്ന് എഴുതുന്നതിന് പകരം 4B10 1036 എന്ന് memory-ൽ എഴുതിയത്. ഇത് കാണുമ്പോൾ തന്നെ '1036 എന്ന memory location-ലേക്ക് ചാടിക്കോ' എന്നാണ് Instruction എന്ന് system-ന് മനസ്സിലായിക്കൊള്ളും. ☺ }

ഇനി, Consider Figure 2.7 (b)



ഇവിടെ pgm load ചെയ്തത് 5000 എന്ന memory location-ൽ ആണ്. So കഥ മാറും! എങ്ങനെ എന്നല്ലേ? ചമയാം.... ☺

നേരത്തെ (in fig (a)) +JSUB RDREC-ന്റെ object code 4B10 1036 ആയിരുന്നു. അതായത്; Jump to the location 1036. ഇവിടെ 1036-ലേക്ക് ചാടിയിട്ട് കാര്യമില്ല! B'coz RDREC is in the location 6036. So, the object code of +JSUB RDREC now becomes 4B10 6036

ie 4B10 1036 + ← address field in the instruction
5000 ← starting address of pgm
4B10 6036

{ നമ്മുടെ instruction-ന്റെ last 4 bits ആണ് address field. ie here, 1036 in 4B10 1036. ഈ 1036-ന്റെ കൂടെയാണ് 5000 add ചെയ്തത് }

ഇതുപോലെ തന്നെ 7420 എന്ന location-ൽ object pgm load ചെയ്യുമ്പോൾ ഉണ്ടാവുന്ന changes ആണ് Figure 2.7 (c)

4B10 1036 +
7420
4B10 8456

ഇതും ~~അങ്ങനെ~~ തന്നെ അല്ലേ +JSUB RDREC-ന്റെ object code ???

ഈ ഉദ്ദേശം നേരം നമ്മൾ discuss ചെയ്തത് എന്താ? Object pgm-നെ വേറൊരു location-ൽ load ചെയ്യുന്നു. ഇതു തന്നെയാ **Program Relocation**. ഈ Program relocation കാരണം object pgm-ൽ ചില changes വരും എന്നും നമ്മൾ മനസ്സിലാക്കി. ഏതൊക്കെ കണ്ടില്ലേ; ഒരോ location മാറുന്നതിന് അനുസരിച്ച് +JSUB RDREC എന്ന instruction-ന്റെ object code മാറി. ഈ മാറ്റം Assembler-ന് predict ചെയ്യാൻ പറ്റില്ല. B'coz Pgm എവിടെ load ചെയ്യുന്നു എന്നതിനെ depend ചെയ്തിരിക്കും. അപ്പോ എവിടെയാണ് മാറ്റം വരുന്നത് എന്ന് Assembler ~~=~~ loader-നോട് പറയും. Through **Modification Reword**. {ഇവിടെ modification വന്ന instruction; അതായത് മാറ്റം വന്ന instruction +JSUB RDREC ആണ്}. അങ്ങനെ നമ്മുടെ object program-ൽ Header, Text and End record-നെ കൂടാതെ ഒരു പുതിയ reword കൂടി വന്നു. Modification Reword; starting with 'M': (~~No~~ Pgm relocation വരി) Pgm ൽ എന്നെങ്കിലും modification ഉണ്ടെങ്കിൽ മാത്രം Modification reword use ചെയ്താൽ മതി.)
Consider **Figure 2.8**. ഇവിടെ 3 modification rewords ഉണ്ട്.

M^000007^05
M^000014^05
M^000027^05

Take the 1st one. M^000007^05

- M → modification reword
- 000007 → Starting ~~add~~ location of the address field to be modified
- 05 → length of address field to be modified (in half bytes)

Refer fig. 2.6.
+JSUB RDREC എന്ന instruction 0006-ൽ ആണ്; and starting address is 0000. So 0007 എന്നതാണ് ~~mod~~ address-il ആണ് modify ചെയ്യണ്ട സംഭവം.

Similarly +JSUB WRREC is in 0013 and again it is in 0026.

ഇനി '05' എന്താണെന്ന് നോക്കാം. See, +JSUB RDREC instruction is in Format 4. ഇവിടെ 20 bits ആണ് address field-ന് വേണ്ടിയുള്ളത്; ie (4+4+4+4+4) bits; ie 5 half bytes! {1 byte = 8 bits}

So Modification Record എന്തിനായി വരുമെന്ന് ഇപ്പോൾ മനസ്സിലായില്ലേ?

OK. ഇതുവരെയുള്ള Machine 'Dependent' assembler Features.

ഇനി നമുക്ക് **Machine Independent Assembler features** എന്തൊക്കെയാണെന്ന് നോക്കാം. ഇവ ഈ features (independent) എല്ലാ assemblers-നും സ്വീകരിക്കാവുന്നതാണ്. ie, SIC/XE ആയാലും VAX ആയാലും CRAY ആയാലും ഇനി പറയാൻ പോകുന്ന features അതിലൊക്കെ കാണാം.

First one is **LITERALS**

Consider Fig 2.6 (line no: 45)

```
001A    ENDFIL  LDA   EOF      {instruction is to load
                                 the character constant
                                 EOF to accumulator}
```

(line no: 80)

```
002D    EOF     BYTE  C'EOF'
```

Now Consider Fig 2.10. മുകളിൽ പറഞ്ഞിരിക്കുന്ന 2 statements-നും കൂടി താഴെയാണ് ഇവിടെ കൊടുത്തത്.

(line no: 45)

```
001A    ENDFIL  LDA   =C'EOF'.
```

Here, the value of the constant operator is written as a part of the instruction that uses it. {ഇവിടെ constant operator ഏതാ? 'EOF'. Fig 2.6-ൽ EOF ഒരു character constant ആണെന്ന് പിന്നീടല്ലേ പറയുന്നുള്ളൂ. (ie line 80). പക്ഷേ

ഇവിടെ, നമ്മൾ EOF use ചെയ്ത Instruction-ൽ തന്നെ പറയും 'It is a character constant'. മനസ്സിലായോ? ഇങ്ങനെയുള്ള 'operands' നെയാണ് നമ്മൾ 'literals' എന്ന് പറയുന്നത്. Literal ആണെന്ന് കാണിക്കാൻ ഉപയോഗിക്കുന്ന symbol ആണ് =.
{Here, literal is =C'EOF'}.

ഒരു Pgm-ൽ നമ്മൾ use ചെയ്ത literal operands-നെ എല്ലാം കൂടി ചേർത്ത് ഒരു literal pool ഉണ്ടാക്കും. {ഒന്നോ; അതിലധികമോ}. എന്നിട്ട് Pgm-ന്റെ END statement കഴിഞ്ഞ് place ചെയ്യും. ie All of the literal operands used in a program are gathered together into one or more literal pools; and they are placed immediately following the END statement. ഇനി, ബാക്കി സംഭവം പറയുന്നതിന് മുമ്പ് 'LTORG' എന്താണെന്ന് നോക്കാം. LTORG is an assembler directive. When the assembler encounters a LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG or the beginning of the program. This literal pool is placed in the object program at the location where the LTORG directive was encountered. കാര്യം പിടികിട്ടിയോ?? അതായത്; Simple ആയിട്ട് പറഞ്ഞാൽ; oru LTORG directive-ന് തൊട്ട് താഴെ Pgm-ൽ അതുവരെ use ചെയ്തിട്ടുള്ള literal operands എല്ലാം കാണാം. {അതുവരെ means; ഒന്നെങ്കിൽ Pgm-ന്റെ beginning തൊട്ട്; അല്ലെങ്കിൽ തൊട്ടുമുമ്പ് use ചെയ്ത LTORG കഴിഞ്ഞതു മുതൽ}. Refer fig: 2.10

Consider (line no: 93)

LTORG
=C'EOF'.

LTORG directive വരുന്നതുവരെ നമ്മൾ ഒരു literal operand മാത്രമേ ഉപയോഗിച്ചിട്ടുള്ളൂ. ie C'EOF'. (in line no: 45).

ഇനി LTORG ഇല്ലായിരുന്നെങ്കിൽ Pgm-ന്റെ താഴെ END statement കഴിഞ്ഞ് =C'EOF' എഴുതിച്ചേർക്കണം (in the literal pool). ഇവിടെ ഇപ്പോൾ literal pool-ൽ =X'05' എന്ന literal operand മാത്രമേ ഉള്ളൂ.

ഇനി Pass I-ന്റെ സമയത്ത് ഈ literals-നെ യെല്ലാം കൂടി എടുത്ത് 'literal table'-ൽ കൊണ്ടു വയ്ക്കും. LITTAB is the data structure used. {same function as SYMTAB}

അപ്പോൾ literals എന്താണെന്ന് ഒരു idea ആയല്ലോ? ഇനി നമുക്ക് അടുത്ത feature-ലേക്ക് പോകാം.

Symbol Defining Statements!

There are 2 types of Symbol Defining statements. അതിലൊന്ന് 'EQU' use ചെയ്യുന്ന statements; രണ്ടാമത്തെത് 'ORG' use ചെയ്യുന്ന statements. രണ്ടിന്റെയും function എന്താണെന്ന് ഇനി പഠിക്കാം.

Consider line no: 133 of Figure 2.5

+LDT #4096 ആണ് Instruction. ഇതിന്റെ meaning എന്താണ് വെച്ചാൽ; 4096 എന്ന value 'T' register-ലേക്ക് load ചെയ്യുക. {4096 is the maximum length of record that we could read using the subroutine RDREC} പക്ഷേ ഈ statement കണ്ടാൽ നമുക്ക് മനസ്സിലാ- Maximum length ആണെന്ന് നമുക്ക് മനസ്സിലാവില്ല. Suppose ഈ instruction +LDT #MAXLEN എന്നായിരുന്നെങ്കിലോ? എന്നിട്ട് MAXLEN-ലോട്ട് 4096 എന്ന value assign ചെയ്യുക. സംഭവം എളുപ്പമായില്ലേ! ശ, നമുക്ക് MAXLEN എന്ന label-നെ 4096 എന്ന value-യായി 'equate' ചെയ്യണം. അതിന് use ചെയ്യുന്ന assembler directive ആണ് EQU.

```
    +LDT    #MAXLEN
MAXLEN  EQU     4096
```

എന്ന് എഴുതിയാൽ +LDT #4096 എന്നതിന് equivalent ആകും!

EQU - ന്റെ syntax.

symbol	EQU	value

അപ്പോ EQU എന്താണെന്ന് മനസ്സിലായിട്ടുണ്ടല്ലോ? OK ഇനി നമുക്ക് ORG എന്താണെന്ന് നോക്കാം.

ORG refers to 'origin'. അതായത്, Location Counter-ന്റെ ഉള്ളിലുള്ള value-നെ reset ചെയ്യാൻ ഉപയോഗിക്കുന്ന assembler directive ആണ് ORG. Its syntax is

ORG	value

eg)
```
STAB    RESB    1100
        ORG     STAB
```

This statement resets the location counter with the value of STAB.

അങ്ങനെ 2-ാം തരം feature കഴിഞ്ഞു. ഇനി 3rd one; Expressions!

ഇതു പറയണമെങ്കിൽ Absolute terms എന്താണെന്നും Relative terms എന്താണെന്നും അറിഞ്ഞിരിക്കണം.

Absolute terms are constants and they are independent of the starting address whereas **relative terms** depends on the starting address of the program.

Ok. ഇനി come back to Expressions. ഒരു Expression എന്നു പറയുമ്പോൾ അതിലുണ്ടാകുന്ന ഓരോ 'terms' ഉണ്ടാവും. {For eg BUFFER - BUFEND is an expression which contains 2 terms BUFFER and BUFFEND}

Absolute terms മാത്രമുള്ള expression-നെ **Absolute Expression** എന്നും; Relative terms മാത്രമുള്ള expression-നെ **Relative Expression** എന്നും പറയും! ഇനി; മറ്റൊരു കാര്യം എന്താണെന്നു വെച്ചാൽ Absolute Expression-ന്റെ അകത്ത് Relative terms-നും കാണാം; പക്ഷേ അതിന് ചില restrictions ഉണ്ടാവും. അതായത്, absolute

expression-ൽ ഉള്ള relative terms 'pair' ആയിട്ടിരിക്കണം; കൂടാതെ അതിന്റെ signs opposite ആയിരിക്കണം. മനസ്സിലായില്ല അല്ലേ? 😊 പറഞ്ഞു തരാം... Consider Figure 2.10 (line no: 107)

```
MAXLEN  EQU  BUFEND - BUFFER
```

ഈ statement-ലെ expression Absolute ആണോ അതോ Relative ആണോ? BUFFER-ഉം BUFEND-ഉം Relative terms ആണ്. കാരണം അതിന്റെ address (0036 and 1036; from line no: 105 and 106) starting address-നെ depend ചെയ്യും. If starting address is 0000, then

```
0036  BUFFER
1036  BUFFEND
```

If starting addres is 5000, then

```
5036  BUFFER
6036  BUFFEND
```

അത് അറിയാമല്ലോ, അല്ലേ? Ok. So normal രീതിയിൽ നമ്മുടെ Expression Relative ആയിരിക്കണം. പക്ഷേ 'It is an Absolute Expression!' 😊 അതിന്റെ Reason എന്താണെന്ന് വെച്ചാൽ; see, starting address എന്തായാലും BUFEND − BUFFER എന്നത് ഒരു constant value അല്ലേ? ie 1000.

```
{ 1036 - 0036 = 1000
  6036 - 5036 = 1000 }
```

That's y we say that

```
MAXLEN  EQU  BUFEND - BUFFER
```

is an absolute expression.

ഇവിടെ Relative terms നമുക്ക് pair ആയിട്ട് ആയിരുന്നു. (2 എണ്ണം). അതുപോലെ അതിന്റെ signs-ഉം opposite ആയിരുന്നു.

```
{ + BUFFEND
  - BUFFER }
```

ഈ പ്രശ്നം കുറച്ച് clear ആയിട്ടുണ്ടല്ലോ? ഇനി 4-ാം തരം

feature ചർച്ചക്കാം. **Program Blocks!**

നമ്മുടെ Pgm-ൽ similar ആയിട്ടുള്ള statements ഉണ്ടെങ്കിൽ അതിനെ എല്ലാം കൂടി group ചെയ്ത് എഴുതാം. ഇങ്ങനെ എഴുതുമ്പോൾ ആ statements എല്ലാം കൂടി Pgm-ന്റെ ഒരു 'Block' ആയി. Pgm blocks-നെ indicate ചെയ്യാൻ ഉപയോഗിക്കുന്ന assembler directive ആണ് USE

For eg) Consider the Figure 2.12. (line no: 92) USE CDATA. അതിന് തൊട്ടുതാഴെ രണ്ടു RESW statements-ഉം ഉണ്ട്. അതായത്, നമ്മുടെ Pgm-ലെ RESW statements എല്ലാം കൂടി group ചെയ്ത് CDATA എന്ന block ഉണ്ടാക്കി. അതുപോലെ തന്നെ line no: 103-ൽ 'CBLKS' എന്നൊരു Block-ഉം start ചെയ്യുന്നുണ്ട്. അങ്ങനെ പ്രത്യേകമായി 2 pgm. blocks ആണ് ഉള്ളത്. പക്ഷേ actually നമ്മുടെ Pgm-ൽ 3 blocks ഉണ്ട്. CDATA, CBLKS and .... default! :)

പറഞ്ഞിട്ടുള്ള 2 blocks-ലും ഉൾപ്പെടാത്ത statements എല്ലാം കൂടി ചേർന്നതാണ് default block.

ഇനി; Assembler എങ്ങനെയാണ് Program blocks-നെ handle ചെയ്യുന്നത് എന്ന് നോക്കാം.

'ഓരോ Pgm block-നും seperate ആയിട്ട് location counter ഉണ്ടായിരിക്കും.' ഇതുവഴി നമ്മൾ source Pgm-നെ ഒന്നു block ആയിട്ടു ആയിരുന്നു consider ചെയ്തത്. ഇപ്പോൾ, default block-ഉം ഓരോ location counter വീതമാണ് ~~ഉണ്ടായിരുന്നു~~ ഉണ്ടായിരുന്നതെന്ന്. ഒരു Pgm block-ന്റെ execution start ചെയ്യുമ്പോൾ അവിടുത്തെ LOCCTR-ന്റെ value is initialized to '0', and previous block-ലെ LOCCTR-ന്റെ value save ചെയ്തും വെയ്ക്കും. ഇനി, previous block-ലേക്ക് എപ്പോഴെങ്കിലും തിരിച്ചു പോകുവാണെങ്കിൽ save ചെയ്ത value restore ചെയ്യും.

പിന്നെ, ഇവിടെ labels SYMTAB-ൽ enter ചെയ്യുമ്പോൾ ആ label ഏതു Block-ൽ ആണ് എന്നു കൂടി specify ചെയ്യും. (Through

Block name or block number.

{ line no: 92 and 103 നോക്കിയാൽ u can see that loc ~~address~~ is 0000 }

ഇനി, ശ്രദ്ധിക്കേണ്ട മറ്റൊരു കാര്യം എന്താണ് വെച്ചാൽ Pgm blocks one by one ആയിട്ട് ആയിരിക്കും memory-ൽ load ചെയ്യുന്നത്. ഈ table കാണുമ്പോൾ ഞാൻ പറഞ്ഞത് എന്താണെന്ന് clear ആകും.

Blockname	Block no:	Address	length
default	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

0066 + 000B = 0071

default block-ന്റെ starting address is 0000 അതിന്റെ length 0066. അതായത് default block occupy ചെയ്യുന്നത് from 0000 to 0065. എന്നുവച്ചാൽ അടുത്ത block വരുന്നത് 0066 എന്ന address-ൽ. CDATA-ടെ starting address 0066 വന്നത് എങ്ങനെയാ എന്ന് ഇപ്പൊ, കണ്ടില്ലേ? ☺ ഇതുപോലെ തന്നെയാണ് CBLKS-ന്റെ case-ഉം! 0066 മുതൽ 0070 വരെ CDATA occupy ചെയ്യും. { Since its length is 000B }. അതുകഴിഞ്ഞ് 0071-ൽ CBLKS start ചെയ്യും.

ഇനി 5th feature; **Control Sections!**

നമ്മുടെ Pgm-ലെ statements-നെ ഒന്നു rearrange ചെയ്ത് എഴുതിയാലിരുന്നു Pgm blocks ഉണ്ടാക്കിയിരുന്നത്. എങ്ങനെ വേണമെങ്കിലും rearrange ചെയ്ത് Blocks ഉണ്ടാക്കാം. അതൊക്കെ Programmer-ന്റെ ഇഷ്ടം! പക്ഷേ Control section എന്നു പറഞ്ഞാൽ അങ്ങനെയല്ല. അതൊരു Independent 'module' ആയിരിക്കും; and has its own identity. For eg) നമ്മുടെ Pgm-ലെ RDREC എന്നു പറയുന്നത് ഒരു Control section ആണ്. ☺

{ RDREC is used to read ~~char~~ record and store in BUFFER, ഓർമ്മ? അതായത് RDREC-ന് സ്വന്തമായി ഒരു ജോലിയുണ്ട്; ഒരു Identity ഉണ്ട്! ഈ RDREC വേറെ എവിടെ പോയാലും;

അൽപ്പ മുമ്പ് അതെങ്ങിൻ പോലും അതിന്റെ ജോലി ചെയ്തോളും. മനസ്സിലായോ ??
So, ഒരു Pgm-ലെ subroutines-നെയും logical subdivisions-നെയും ആണ് നമ്മൾ control section ആയിട്ട് കാണുന്നത്. ഇവിടെ use ചെയ്യുന്ന assembler directive ആണ് CSECT. For eg) Consider Figure 2.15. line no 109 കണ്ടോ? RDREC CSECT. This means that RDREC is a control section. Similarly in line no 193, it is said that WRREC is a control section. ഇപ്പോ control section-ന്റെ concept clear ആയിട്ടല്ലേ? OK. മനസ്സിലാകേണ്ട വേറൊരു കാര്യം എന്താണ് വെച്ചാൽ: 'Each control section can be loaded and relocated independently of the others.' അതായത് RDREC എന്ന control section load ചെയ്ത് കഴിഞ്ഞ ഉടനേ തന്നെ WRREC load ചെയ്യണം എന്ന് നിർബന്ധമില്ല. പിന്നെ എപ്പഴെങ്കിലും load ചെയ്താലും മതി! 😊 പക്ഷേ അപ്പോൾ ഒരു പ്രശ്നം ഉണ്ട്. എന്താണെന്ന് പറയാമോ???? OK ഞാൻ തന്നെ പറയാം. 🙂
Suppose നമ്മൾ main program മാത്രം load ചെയ്തു. Execution time-ൽ +JSUB RDREC എന്ന് കണ്ടു. പക്ഷേ എങ്ങോട്ട് Jump ചെയ്യും?! B'coz RDREC ഇനിവരെ നമ്മൾ load ചെയ്തിട്ടില്ല.
So, record read ചെയ്യാത്ത ~~code~~ code RDREC എന്ന control section-ൽ ആണെങ്കിൽ, RDREC-ൽ പോയാൽ മാത്രമേ code (definition) കിട്ടൂ എന്നും ~~അപ്പോ~~ Pgm-ൽ നേരത്തെ തന്നെ പറയണം. അതിനു വേണ്ടി use ചെയ്യുന്ന assembler directive ആണ് EXTREF. Check line no: 7 EXTREF RDREC, WRREC കണ്ടോ?? This means that ഈ രണ്ട് symbols-ഉം (RDREC & WRREC) നമ്മൾ ഈ particular control section-ൽ ഉപയോഗിച്ചിട്ടുണ്ട്; പക്ഷേ അത് എന്താണെന്ന് പറയുന്നത് വേറെ control section-ൽ ആണ്.

അർത്ഥം മനസ്സിലായോ??

Similarly; Check line no: 122 (Control section is RDREC)

EXTREF BUFFER, LENGTH, BUFEND

ഇതിന്റെ meaning എന്താ? BUFFER, LENGTH, BUFFEND എന്നീ symbols നമ്മൾ ഇന്ന particular control section-ൽ (RDREC-ൽ) ഉപയോഗിച്ചിട്ടുണ്ട്. പക്ഷേ ഇതൊന്നുമല്ല എന്താണെന്ന് പറയുന്നത് വേറെ Control section-ൽ ആണ് {~~Ok so~~ In our pgm it is in 'main'; line no: 100, 105, 106}. മ്മ, വേറെ control sections-ൽ വേണ്ട ചില symbols-ഉം ഡെഫിനിഷൻ നമ്മൾ main-ൽ പറഞ്ഞു. ഇങ്ങനൊരു സംഭവം കൂടി നടന്നിട്ടുണ്ടെന്ന് നമ്മൾ Pgm-നെ രണ്ടായിട്ട് തന്നെ അറിയിക്കണം. അതിനുവേണ്ടി use ചെയ്യുന്ന assembler directive ആണ് EXTDEF. Check line no: 6

EXTDEF BUFFER, BUFEND, LENGTH.

ഇവ മൂന്നും symbols-ഉം RDREC-ൽ EXTREF ആയിരുന്നില്ലേ?.

ഇപ്പോ technique പിടികിട്ടിയോ? ☺

So, shall I continue?? OK..

അങ്ങനെ ചുരുക്കി രണ്ട് സംഭവങ്ങൾ കൂടി നമ്മുടെ Pgm-ൽ വന്നു; EXTDEF and EXTREF. ഇത് Object Pgm-ൽ എങ്ങനെ specify ചെയ്യും? അതിനുവേണ്ടി use ചെയ്യുന്ന records ആണ് Define Record and Refer Record {Already നമ്മൾ 4 records പഠിച്ചു: Header record, Text Record, Modification record and End record. അതിനിപ്പോ കൂടെ 2 എണ്ണം കൂടി വന്നു; Define record and Refer record. So, we've a total of 6 records!!! മതിയല്ലേ, അല്ലേ? ☺ ചുമ്മാ പറഞ്ഞതാ...} OK, so let us continue with Define & Refer Record.

EXTDEF-ൽ പറഞ്ഞിരിക്കുന്ന symbols

നമ്മൾ 'Define record'-il എഴുതും. അതുപോലെ EXTREF-ൽ പറഞ്ഞിരിക്കുന്ന symbols 'Refer record'-ലും!

Pgm block-ന്റെ case-ൽ നമ്മൾ പറഞ്ഞിരുന്നു; 'each pgm block has seperate location counter. Same way, 'each control section has a seperate location counter (beginning at 0).' അതുപോലെ തന്നെ ഓരോ Control section-നും seperate ആയിട്ടുതിരിക്കും object pgm എഴുതുന്നത്. See Figure 2.17. 3 object pgms അല്ലേ ഉള്ളൂത്? 1st one for COPY, 2nd for RDREC and 3rd for WRREC. (See pgm names in Header record). Then u'll understand).
Header record-ന് തൊട്ടു താഴെയാണ് Define record ഉള്ളത് (starting with D)

D^BUFFER^000033^BUFEND^001033^LENGTH,000020

ie D^name of symbol^its address

അങ്ങനെ എത്ര symbols ഉണ്ടോ അത്രയും!
തൊട്ടു താഴെയാണ് Refer record എഴുതുന്നത്.

R^RDREC^WRREC

ഇവിടെ address ഒന്നും വേണ്ട. Just name മാത്രം specify ചെയ്താൽ മതി.
ബാക്കി എല്ലാം പഴയതു പോലെ തന്നെ! Text record എഴുതുക; Modification ഉണ്ടെങ്കിൽ modification record എഴുതുക; Then End record!! Thats all!
അങ്ങനെ Assembler features-ഉം കഴിഞ്ഞു! ☺
ഓരോ Topics-ഉം കുറെ കൂടി Deep ആയിട്ട് Text-il പറഞ്ഞിട്ടുണ്ട്. അതുകൂടി നോക്കുക.

ഓ, ഇനി നമുക്ക് രണ്ടു Topics-കൂടി പഠിക്കാനുണ്ട്; One-pass Assembler and Multi-Pass Assemblers {MASM & SPARC text-ൽ ഉള്ളത് അതുപോലെ പഠിക്കുക. അതിനുകാരണം (പ്രത്യേകിച്ച് ഒന്നും പറയാനില്ല! ☺ }

അപ്പോ **One-pass Assembler** തുടങ്ങാം ? Ok!
ഈ പേര് കേൾക്കുമ്പോൾ തന്നെ ഒരു കാര്യം നമുക്ക് മനസ്സിലാക്കാൻ പറ്റും. അതായത്; ഇവിടെ ഒറ്റ ഒരു pass-ൽ തന്നെ എല്ലാ പരിപാടികളും കഴിയും. നേരത്തെ നമ്മൾ പറഞ്ഞ ഒരു Problem ആയിരുന്ന forward reference. അത് പരിഹരിക്കാനായിരുന്നു 2nd pass. അല്ലേ? ഈ Assembler-ൽ ഒരു Pass മാത്രമേ ഉള്ളൂ എന്നും പറയുന്നു. അപ്പോ ഇവിടെ എങ്ങനെയാ forward reference പരിഹരിക്കുന്നത് ?? അങ്ങനൊരു സംശയം തോന്നിയില്ലേ? തോന്നണം! 😊 } പറഞ്ഞുതരാം...

See, ഒരു Pgm-ൽ symbols ഒക്കെ use ചെയ്ത് കഴിഞ്ഞ് define ചെയ്യുമ്പോഴല്ലേ forward reference problem ഉണ്ടാകുന്നത്. So അത് avoid ചെയ്യാൻ എന്താ ചെയ്ക ?? Symbols എല്ലാം ആദ്യമേ തന്നെ define ചെയ്യണം! Simple !! 😉

Refer **Figure 2.18**. Symbols എല്ലാം ആദ്യമേ തന്നെ define ചെയ്തിട്ടില്ലേ ? (line no: 1 to 6) പക്ഷേ എന്നൊക്കെ പറഞ്ഞാലും ചില forward reference നമുക്ക് ഇതുപോലെ solve ചെയ്യാൻ പറ്റില്ല. For eg, Consider line no: 15
JSUB RDREC.
RDREC ഉള്ളത് 2030 എന്ന location-ൽ ആണ്. അതെടുത്ത് മുകളിൽ കൊണ്ടുവെയ്ക്കാൻ Pgm-ന്റെ logic തന്നെ മാറിപ്പോകും! അല്ലേ?? So ഈ situation എങ്ങനെയാ handle ചെയ്യുന്നത് എന്ന് പറയണമെങ്കിൽ first you've to understand that there are 2 main types of One-pass Assembler.

i) **load-and-go assembler** : ഇവിടെ object code memory-ൽ ആണ് generate ആകുന്നത്. (not in secondary device as in usual case). Here, no object program is written out

and no loader is needed.
{They produce object code directly in memory for immediate execution'}
2nd type produces object program for later execution. അതിന് പ്രത്യേകിച്ച് പേരൊന്നും ഇല്ല!
ഓ, ഇനി നമുക്ക് load-and-go assembler-ന്റെ case എടുത്ത് forward reference എങ്ങനെയാ solve ചെയ്യുന്നത് എന്ന് പഠിക്കാം. {That is more easy to understand}. ഇവിടെ assembler source pgm fine by line ആയിട്ട് scan ചെയ്ത് ഓരോ line-ന്റെയും object code generate ചെയ്യുവാണല്ലോ. ഓ അങ്ങനെ scan ചെയ്യുമ്പോൾ എവിടെയെങ്കിലും forward refered ആയിട്ടുള്ള ഒരു symbol കണ്ടാൽ; ആ statement-ലെ object code-ന്റെ operand address part അങ്ങ് omit ചെയ്യും. അതായത് വിട്ടുകളയും! Just see Figure 2.19 (a). Figure 2.18-ലെ pgm memory-il എങ്ങനെയാ എന്നാണ് കാണിച്ചിരിക്കുന്നത് {കുറച്ച് portion മാത്രമേ കാണിച്ചിട്ടുള്ളൂ}

Memory address	Contents
1000	454F4600 0003 0000 00xxxxxx xxxxxxxx

(line no: 1 in fig 2.18) → 454F4600; line no: 2 → 0003; line no: 3 → 0000

ഈ മൂന്ന object codes കഴിഞ്ഞ് 'x' marks reserve ചെയ്തു വച്ചിരിക്കുന്ന memory space ആണ് (for later use). ഇത് കഴിഞ്ഞ് വീണ്ടും object code start ചെയ്യുന്നത് line no: 10-ന്റെ ആണ് ie, 141009. ഇത് ~~ഈ~~ ഈ object code memory-il 2000-ലും 2010-ലും ആയിട്ടാണ് ഉള്ളത്.

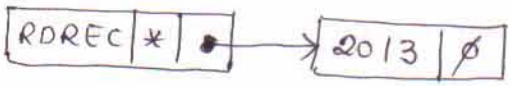
xxxxxx 14 1009 48-- --

141009

ഇത് കഴിഞ്ഞാണ് നമുക്ക് forward reference വരുന്ന

statement.

ie JSUB RDREC (in line no: 15)
RDREC-ന്റെ address 2038 (in line no. 125)
ഇപ്പോൾ memory-ൽ ഈ 2038 എന്തൊക്കെ space blank ആണ്? (represented by -) അതാ പറഞ്ഞത് forward reference ഉള്ളതിന്റെ address part നമ്മൾ omit ചെയ്യും; എന്നിട്ട് ആ symbol SYMTAB-il enter ചെയ്യും as an undefined symbol (indicated by *). Fig. 2.19 (a)-ൽ നമ്മ right side-ൽ symbol table-ും കാണിച്ചിട്ടുണ്ട്. അതിന്റെ 2nd row നോക്കൂ.



RDREC-ന്റെ value ~~ആ~~ എപ്പോഴെങ്കിലും കിട്ടുമ്പോൾ assign ചെയ്യുന്ന address ആണ് link ചെയ്ത് കാണിച്ചിരിക്കുന്നത്. {Here it is 2013. 2012 il JSUB store ആകും. (ie 48). അത് കഴിഞ്ഞ് 2013 & 2014-ൽ ആണ് RDREC-ന്റെ address ആയ 2038 പോകുന്നത്. 20 goes to address 2013 and 38 goes to 2014} See the symbol table in Figure 2.19 (b). 2nd row-il RDREC-ന്റെ value enter ചെയ്തിട്ടില്ല? ഇനി WRREC-ൽ രണ്ടു link വന്നതു കണ്ടോ? It means that whenever we get the location of WRREC, it should be given to both the addresses. (ie to 201F as well as 2031)
ഇതൊരുപാടു സംസാരം! ഇനി **Multi-pass Assembler**

അതിലധികം passes-ൽ കൂടി object code generate ചെയ്യുന്ന Assembler ആണ് multipass assembler. Pgm-ൽ കുറെ 'EQU' statements വരുമ്പോഴാണ് multipasses ആവശ്യമായി വരുന്നത്.

For eg) Consider the statements,

```
ALPHA   EQU   BETA
BETA    EQU   DELTA
DELTA   RESW  1
```

ഇവിടെ Pass 1 കഴിയുമ്പോൾ DELTA-ടെ value കിട്ടും. Pass 2-ൽ Delta value Beta-യുമായി equate ചെയ്തു. അങ്ങനെ Beta-ടെ value-ഉം കിട്ടി. പക്ഷേ അപ്പോഴും Alpha വെറുതെ നിൽക്കുവാ. Hence we need a 3rd pass inorder to equate the value of beta to alpha. ഇപ്പൊ OK ആയില്ലേ? ശരി!

Text-il ഒരു example പറഞ്ഞിട്ടുണ്ട്; with diagram. അതെന്താണെന്ന് നോക്കാം.

Take Figure 2.21

```
HALFSZ   EQU   MAXLEN/2
MAXLEN   EQU   BUFEND-BUFFER
PREVBT   EQU   BUFFER-1
BUFFER   RESB  4096
BUFEND   EQU   *
```

ഇത്രയും statements ആണ് തന്നിട്ടുള്ളത്.
{* means present value of LOCCTR}. ഈ statements ശ്രദ്ധിച്ചാൽ നിങ്ങൾക്ക് മനസ്സിലാവും; 1st pass കഴിയുമ്പോൾ BUFFER-ന്റെയും BUFFEND-ന്റെയും value കിട്ടും; ഈ രണ്ടു values-ഉം ഉള്ളതുകൊണ്ട് 2nd pass കഴിയുമ്പോൾ MAXLEN-ന്റെയും PREVBT-ന്റെയും values കിട്ടും; അതുപോലെ തന്നെ 3rd pass-ൽ HALFSZ-ന്റെയും!! ഇതൊക്കെയാണ് ഇവിടെ സംഭവിക്കുന്നത്. ഇനി diagrams ഒന്നു നോക്കാം. നാലഞ്ചു പടം ഉണ്ട് കേട്ടോ! സംഭവം ഒന്നുമില്ല ☺ അതു ഞാൻ പറഞ്ഞു കഴിയുമ്പോൾ മനസ്സിലാകും.

തുടങ്ങാം? OK... Take Fig 2.21 (b)

HALFSZ	&1	MAXLEN/2	∅

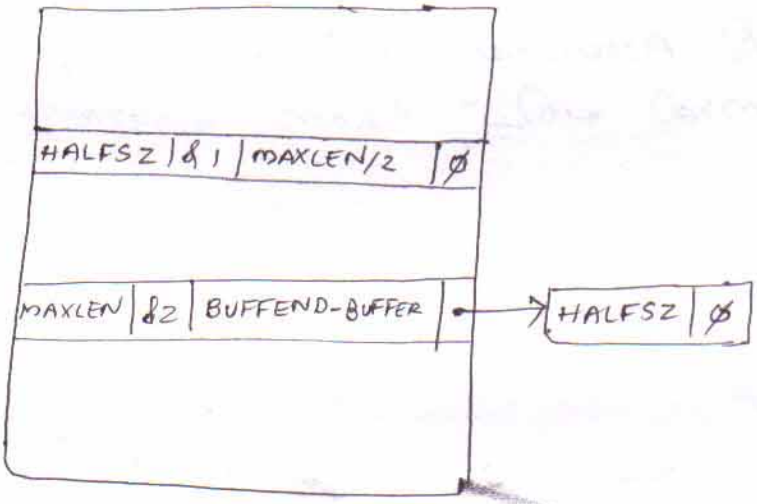
ഈ കാണിച്ചിരിക്കുന്നത് നമ്മുടെ 1st statement ആണ്

അതാണ് HALFSZ EQU MAXLEN/2
#1 means ഇവിടെ ഒരു undefined symbol ഉണ്ട്. ആ MAXLEN!

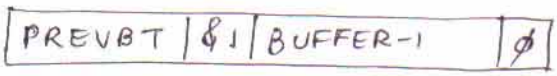
MAXLEN undefined ആണെന്നും, അത് കിട്ടിയാൽ HALFSZ-ന്റെ value കിട്ടും. എന്നാണ് ഇതിന്റെ meaning.
Now we go to the 2nd statement

MAXLEN EQU BUFEND - BUFFER.

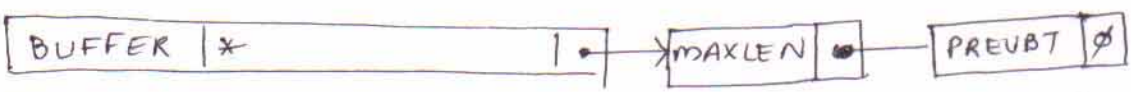
ഇത് represent ചെയ്തിരിക്കുന്നത് Figure 2.21(c)-ൽ ആണ്. കണ്ടോ??



ഇതിൽ 2 undefined statements ഉണ്ട്; BUFFEND and BUFFER (Text-ൽ * ഇട്ട് കാണിച്ചിട്ടുണ്ട്). അതുകൊണ്ടാണ് #2 വന്നത്.
ഇതുപോലെ തന്നെ 3rd statement കൂടി represent ചെയ്യുമ്പോൾ വരുന്ന diagram ആണ് 2.21(d)
ഇവിടെ PREVBT എന്നൊരു symbol കൂടി കയറി.



Now,



BUFFER-ന്റെ ~~#~~ value കിട്ടുമ്പോൾ MAXLEN-ും PREVBT-ും കണ്ടുപിടിക്കാൻ ഉപയോഗിക്കാം. എന്നാണ് ഇതിന്റെ meaning.

Now Fig 2.21 (e) - ഇവിടെ BUFFER-ന്റെ value കിട്ടിക്കഴിഞ്ഞു. {1034 എന്ന് വെറുതെ Assume ചെയ്തതാണ്}. So, obviously PREVBT-ക്കും കിട്ടി.
{Since PREVBT = BUFFER - 1}

Fig 2.21 (f) - ൽ BUFEND-ന്റെ value-ക്കും കിട്ടി.
So MAXLEN calculate ചെയ്തു. (BUFEND - BUFFER)
അതുകഴിഞ്ഞ് HALFSZ (MAXLEN/2)
അങ്ങനെ after multi-passes; എല്ലാ symbols-ന്റെയും values നമുക്ക് കിട്ടി.

☺ കഴിഞ്ഞു! ഇതാ ഉള്ളൂ Assembler... കണ്ടോ...
എത്ര simple ആ!! പാവമല്ലേ നമ്മുടെ Assembler??
ഇനി Assembler-നെ പേടിക്കണ്ടാ... അതും!!
നന്നായി പഠിച്ച് Exam എഴുതുക.. All the best!