System Programming
Chapter Two: Assemblers

Haymanot F.(M.Sc.)

Debre Markos University
School of computing
Department of Software Engineering

2013 E.C.

CONTENTS

- Basic assembler functions
  - A simple SIC assembler
  - Assembler algorithm and data structures
  - Machine dependent assembler features
    - Instruction formats and addressing modes
    - Program relocation
  - Machine independent assembler features
    - Literals - Symbol-defining statements - Expressions
- Assembler design option
  - One pass assemblers and Multi pass assemblers

Basic assembler functions

- Assembler : is a program which translate a source code written by assembly language in to machine language code.
- Fundamental functions
  - Assign machine addresses to symbolic labels used by the programmers
  - Translate mnemonic operation codes to their machine language equivalents
- Machine dependency
  - Depend heavily on the source language it translates and the machine language it produces
  - Ex. different machine instruction formats and codes

Advantage & disadvantage of assembly language

Advantage:

- Increase the efficiency of programmer: *since we can not write the object code using machine language*
- Programmer has flexibility in writing programs

Customized to the specific computer:

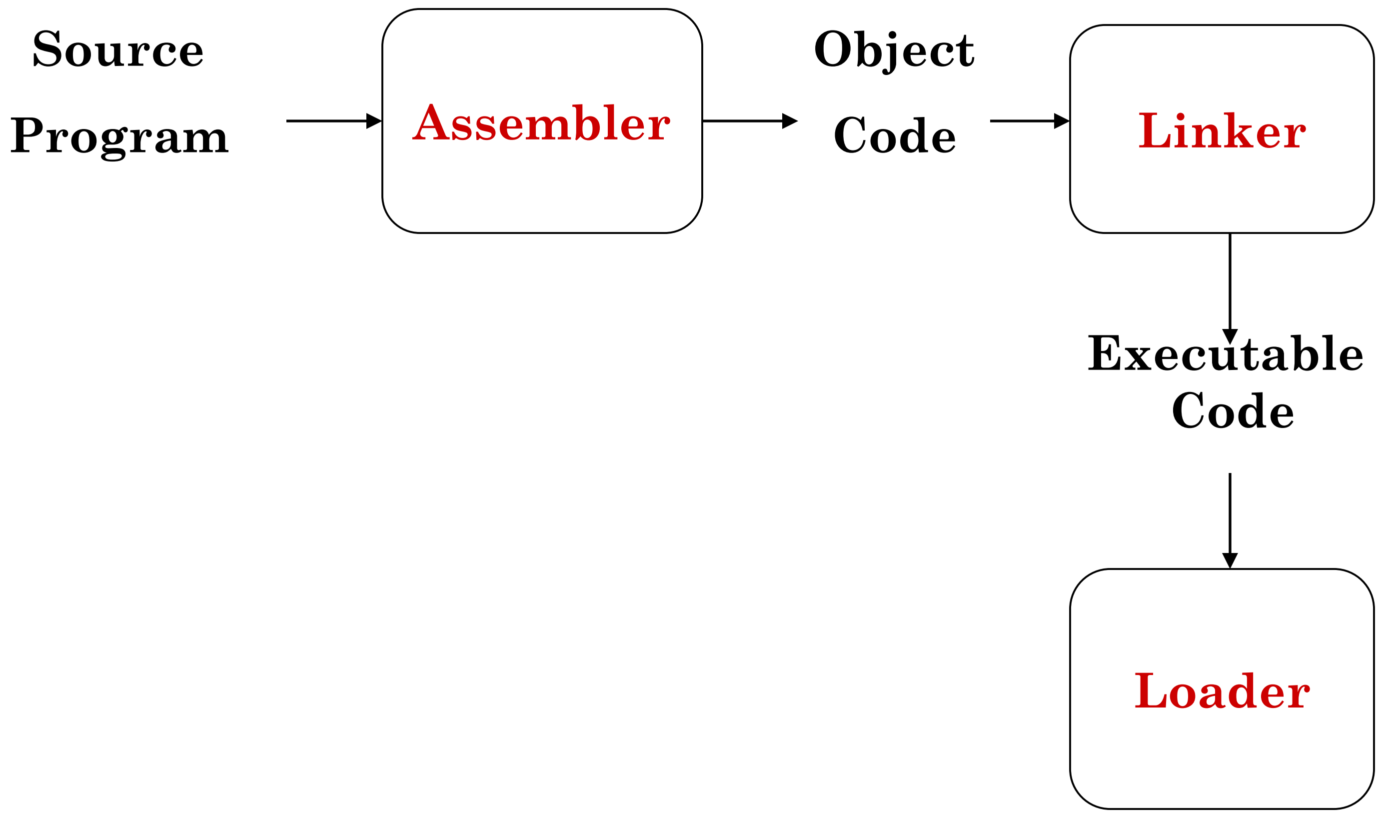
  - *different computer has different instruction format*
  - *The programmer write different programs for each*

- Execute faster than high-level languages
- Highly used in system software development

Disadvantage:

- Costly in terms of programmer time
- Difficult to read and debug, and
- Difficult to learn

Role of Assemblers



ASSEMBLER DIRECTIVES/COMMANDS

- Also called Pseudo-Instructions
- Assembler directives
  - Not translated into machine instructions
  - Provides instructions to the assembler
- Basic assembler directives
  - START
    - Specify name and starting address for the program
  - END
    - Indicate the end of the source program, and (optionally) the first executable instruction in the program

Assembler Directives…

- BYTE
  - Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant
- WORD: Generate one-word integer constant
- RESB: Reserve the indicated number of bytes for a data area
- RESW: Reserve the indicated number of words for a data area
- End of record: a null char(00)
- End of file: a zero-length record

• BYTE & WORD – directs the assembler to generate constants
• RESB & RESW – instructs the assembler to reserve memory locations without generating data values

SIC Assembler program Example

Main program

Specify name and starting address for the program

```
COPY    START   1000      COPY FILE FROM INPUT TO OUTPUT
FIRST   STL     RETADR    SAVE RETURN ADDRESS
CLOOP   JSUB    RDREC     READ INPUT RECORD
        LDA     LENGTH    TEST FOR EOF (LENGTH = 0)
        COMP    ZERO
        JEQ     ENDFIL    EXIT IF EOF FOUND
        JSUB    WRREC     WRITE OUTPUT RECORD
        J       CLOOP     LOOP
ENDFIL  LDA     EOF       INSERT END OF FILE MARKER
        STA     BUFFER
        LDA     THREE     SET LENGTH = 3
        STA     LENGTH
        JSUB    WRREC     WRITE EOF
        LDL     RETADR    GET RETURN ADDRESS
        RSUB              RETURN TO CALLER
EOF     BYTE    C'EOF'
THREE   WORD    3
ZERO    WORD    0
RETADR  RESW    1
LENGTH  RESW    1         LENGTH OF RECORD
BUFFER  RESB    4096      4096-BYTE BUFFER AREA
```

End-of-file

End-of-record

Call subroutine

Forward reference

When End-of-file is reached

char

Line numbers are not part of the program. They are for reference only.

SIC Assembler…Example

Comment line

```
        .
        .       SUBROUTINE TO READ RECORD INTO BUFFER
        .
RDREC   LDX     ZERO            CLEAR LOOP COUNTER
        LDA     ZERO            CLEAR A TO ZERO
RLOOP   TD      INPUT           TEST INPUT DEVICE
        JEQ     RLOOP           LOOP UNTIL READY
        RD      INPUT           READ CHARACTER INTO REGISTER A
        COMP    ZERO            TEST FOR END OF RECORD (X'00')
        JEQ     EXIT            EXIT LOOP IF EOR
        STCH    BUFFER,X        STORE CHARACTER IN BUFFER
        TIX     MAXLEN          LOOP UNLESS MAX LENGTH
        JLT     RLOOP             HAS BEEN REACHED
EXIT    STX     LENGTH          SAVE RECORD LENGTH
        RSUB                    RETURN TO CALLER
INPUT   BYTE    X'F1'           CODE FOR INPUT DEVICE
MAXLEN  WORD    4096
        .
```

“<“ means ready

End-of-record-null character

Indexed addressing

Hexadecimal number

SIC Assembler…Example

```
         .
         .        SUBROUTINE TO WRITE RECORD FROM BUFFER
         .
WRREC    LDX      ZERO              CLEAR LOOP COUNTER
WLOOP    TD       OUTPUT            TEST OUTPUT DEVICE
         JEQ      WLOOP             LOOP UNTIL READY
         LDCH     BUFFER,X          GET CHARACTER FROM BUFFER
         WD       OUTPUT            WRITE CHARACTER
         TIX      LENGTH            LOOP UNTIL ALL CHARACTERS
         JLT      WLOOP               HAVE BEEN WRITTEN
         RSUB                       RETURN TO CALLER
OUTPUT   BYTE     X'05'             CODE FOR OUTPUT DEVICE
         END      FIRST
```

Subroutine return point

Assembler's Job

The translation of the source program to the object program requires us to accomplish the following functions:

- Convert mnemonic operation codes to their machine language codes *{Eg: translate STL to 14 (line 10)}*
- Convert symbolic (e.g., jump labels, variable names) operands to their machine addresses *{Eg: translate RETADR to 1033 (line 10)}*
- Use proper addressing modes and formats to build efficient machine instructions
- Translate data constants in the source program into internal machine representations *{Eg: translate EOF to 454F46 (line 80)}*
- Output the object program and provide other information (e.g., for linker and loader)

CON…

- All of above functions can be easily accomplished by sequential processing of source program, 1 line at a time
  - Example Consider line 10

10 1000 FIRST STL RETADR 141033

  - This contains a forward reference, (i.e.,) a reference to a label RETADR that is defined later in the program
  - Line 10 stores the value of L register in RETADR, but RETADR isn't defined yet. It is defined on line 95 only.
  - If we attempt to translate the program line by line, we will be unable to process this statement, because we don't know the address that will be assigned to RETADR
  - Therefore, most assemblers use 2 passes
  - **1**st pass – scan source program for label definitions & assign addresses
  - **2**nd pass – performs most of the actual translation

SIC Example with Object Code

Loc	Source statement			Object code
1000	COPY	START	1000	
1000	FIRST	STL	RETADR	141033
1003	CLOOP	JSUB	RDREC	482039
1006		LDA	LENGTH	001036
1009		COMP	ZERO	281030
100C		JEQ	ENDFIL	301015
100F		JSUB	WRREC	482061
1012		J	CLOOP	3C1003
1015	ENDFIL	LDA	EOF	00102A
1018		STA	BUFFER	0C1039
101B		LDA	THREE	00102D
101E		STA	LENGTH	0C1036
1021		JSUB	WRREC	482061
1024		LDL	RETADR	081033
1027		RSUB		4C0000
102A	EOF	BYTE	C'EOF'	454F46
102D	THREE	WORD	3	000003
1030	ZERO	WORD	0	000000
1033	RETADR	RESW	1	
1036	LENGTH	RESW	1	
1039	BUFFER	RESB	4096	
	.			

SIC Example…object Code

```
        .
        .                SUBROUTINE TO READ RECORD INTO BUFFER
        .
2039    RDREC     LDX      ZERO                  041030
203C              LDA      ZERO                  001030
203F    RLOOP     TD       INPUT                 E0205D
2042              JEQ      RLOOP                 30203F
2045              RD       INPUT                 D8205D
2048              COMP     ZERO                  281030
204B              JEQ      EXIT                  302057
204E              STCH     BUFFER,X              549039
2051              TIX      MAXLEN                2C205E
2054              JLT      RLOOP                 38203F
2057    EXIT      STX      LENGTH                101036
205A              RSUB                           4C0000
205D    INPUT     BYTE     X'F1'                 F1
205E    MAXLEN    WORD     4096                  001000
```

SIC EXAMPLE…OBJECT CODE

```
                .
                .      SUBROUTINE TO WRITE RECORD FROM BUFFER
                .
2061   WRREC    LDX    ZERO           041030
2064   WLOOP    TD     OUTPUT         E02079
2067            JEQ    WLOOP          302064
206A            LDCH   BUFFER,X       509039
206D            WD     OUTPUT         DC2079
2070            TIX    LENGTH         2C1036
2073            JLT    WLOOP          382064
2076            RSUB                  4C0000
2079   OUTPUT   BYTE   X'05'          05
                END    FIRST
```

Examples of translation SIC

- Mnemonic code (or instruction name) → opcode
- Examples:

STL RETADR → 14 10 33

0001 0100	0	001 0000 0011 0011

STCH BUFFER,X → 54 90 39

0101 0100	1	001 0000 0011 1001

Object Program Format

- The assembled program will be loaded into memory for execution.
- The simple object program contains three types of records: Header record, Text record and end record.
- **Header record** contains the starting address and length.
- **Text record** contains the translated instructions and data of the program, together with an indication of the addresses where these are to be loaded.
- **End record** marks the end of the object program and specifies the address where the execution is to begin.

Object Program Format…

- Header record

  Col. 1 H
  Col. 2~7 Program name
  Col. 8~13 Starting address of object program (hex)
  Col. 14~19 Length of object program in bytes (hex)
  Length= ((last address-starting address)+1)

- Text record

  Col. 1 T
  Col. 2~7 Starting address in this record (hex)
  Col. 8~9 Length of object code in this record in bytes (hex)
  Col. 10~69 Object code in hex (2 columns per byte of object code)

- End record

  Col. 1 E
  Col. 2~7 Address of first executable instruction in object program(hex)

OBJECT PROGRAM…

Length (2079 – 1000)+1

H^COPY ^001000^00107A

T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D

T^00101E^15^0C1036^482061^081044^4C0000^454F46^000003^000000

T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F

T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036

T^002073^07^382064^4C0000^05

E^001000

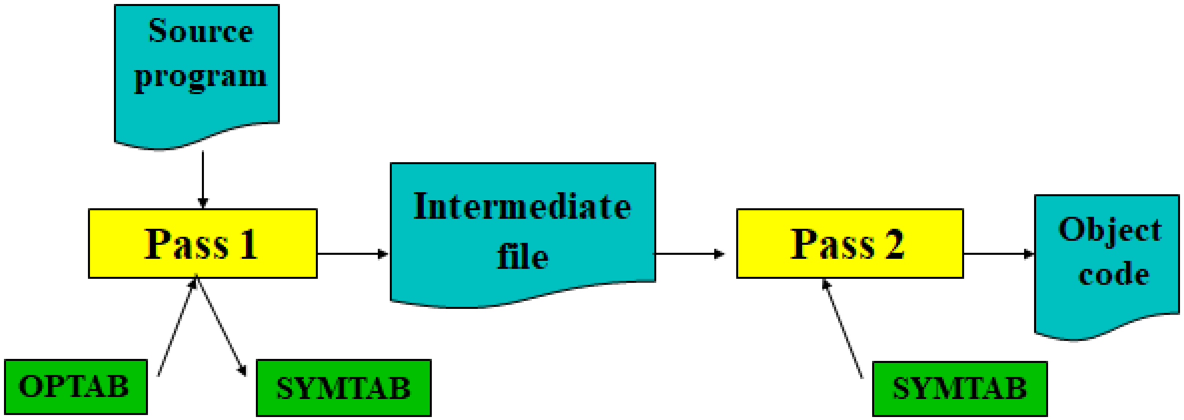
(1032-101E)+1= 15

Length of object code in this record in bytes
(101D-1000)+1
= 1E

Assembler algorithm and data structures

- The simple assembler uses two major internal data structures: the operation Code Table (**OPTAB**) and the Symbol Table (**SYMTAB**).
- LOCCTR: LOCATION COUNTER

- OPTAB looks up mnemonic opcodes & translates them to their machine language equivalents
- SYMTAB stores values (addresses) assigned to labels



OPERATION CODE TABLE (OPTAB)

Content:

- Mnemonic machine code and its machine language equivalent
- also include instruction format, addressing modes ,length etc

Usage:

- Pass 1: used to loop up and validate operation codes in the source program
- Pass 2: used to translate the operation codes to machine language
- In SIC both passes could be done in either pass 1 or pass2 However, for
- SIC/XE, having instructions of different length we use both pass 1 & pass 2

Characteristics:

- Static table - the content will never change
- Contents are not normally added/deleted (predefined)

Example of OPTAB

Mnemonic	MC	Mnemonic	MC
STL	14	TD	E0
JSUB	48	RD	D8
LDA	00	STCH	54
COMP	28	COMPR	A0
JEQ	30	LDL	08
STA	0C	LDT	74
RSUB	4C	LDCH	50
LDX	04	STX	10
J	3C	JLT	38
		JIX	2C

Symbol table (SYMTAB)

Content:

- Include the label name and value (address) for each label in the source program
- May also include flag (type, length) etc

Usage:

- In pass1, labels are entered into SYMTAB as they are encountered in the source program, along with assigned addresses from LOCCTR
- In pass2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions

Characteristics:

- Dynamic table (i.e., symbols may be inserted, deleted, or searched in the table)

EXAMPLE SYMTAB

COPY	1000
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	1024
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039
WRREC	2061

Location Counter (LOCCTR)

- LOCCTR: Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses.
- LOCCTR is initialized to the beginning address mentioned in the START statement of the program.
- After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction.
- Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.
- Reading Assignment
  - Pseudo Code for Pass 1 (SIC)
  - Pseudo Code for Pass 2 (SIC)

Pseudo Code for Pass 1 (SIC)

Pass 1:

```
begin
   read first input line
   if OPCODE = 'START' then
      begin
         save #[OPERAND] as starting address
         initialize LOCCTR to starting address
         write line to intermediate file
         read next input line
      end {if START}
   else
      initialize LOCCTR to 0
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               if there is a symbol in the LABEL field then
                  begin
```

- 1st find starting address of the program
- START – its operand will be the starting address

PSEUDO CODE…

- Whenever we find a label, save it in the symbol table
- Set the error flag if an unrecognized opcode is found OR if a symbol is encountered more than 1 time

```
    search SYMTAB for LABEL
    if found then
        set error flag (duplicate symbol)
    else
        insert (LABEL,LOCCTR) into SYMTAB
end {if symbol}
search OPTAB for OPCODE
if found then
    add 3 {instruction length} to LOCCTR
else if OPCODE = 'WORD' then
    add 3 to LOCCTR
else if OPCODE = 'RESW' then
    add 3 * #[OPERAND] to LOCCTR
else if OPCODE = 'RESB' then
    add #[OPERAND] to LOCCTR
```

PSEUDO CODE…

```
                else if OPCODE  = 'BYTE' then
                    begin
                        find length of constant in bytes
                        add length to LOCCTR
                    end {if BYTE}
                else
                    set error flag (invalid operation code)
            end {if not a comment}
        write line to intermediate file
        read next input line
    end {while not END}
  write last line to intermediate file
  save (LOCCTR - starting address) as program length
end {Pass 1}
```

PSEUDO CODE FOR PASS 2 (SIC)

Pass 2:

```
begin
   read first input line {from intermediate file}
   if OPCODE = 'START' then
      begin
         write listing line
         read next input line
      end {if START}
   write Header record to object program
   initialize first Text record
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               search OPTAB for OPCODE
```

- Write the HEADER

Pseudo Code…

```
if found then
    begin
        if there is a symbol in OPERAND field then
            begin
                search SYMTAB for OPERAND
                if found then
                    store symbol value as operand address
                else
                    begin
                        store 0 as operand address
                        set error flag (undefined symbol)
                    end
            end {if symbol}
        else
            store 0 as operand address
        assemble the object code instruction
    end {if opcode found}
else if OPCODE = 'BYTE' or 'WORD' then
    convert constant to object code
```

Pseudo Code…

```
            if object code will not fit into the current Text record then
                begin
                    write Text record to object program
                    initialize new Text record
                end
            add object code to Text record
        end {if not comment}
    write listing line
    read next input line
  end {while not END}
write last Text record to object program
write End record to object program
write last listing line
end {Pass 2}
```

Machine dependent assembler features

- SIC/XE
- Instruction formats and addressing modes
- Program relocation

SIC/XE Assembler

What's new for SIC/XE?

- more addressing modes and instruction format than SIC.
- program relocation

- Register-to-register instructions are used to improve execution speed.
  - Fetching a value stored in a register is much faster than fetching it from the memory, because they are shorter & don't require another memory reference.
  - In line 150, COMP ZERO is changed to COMPR A,S
  - Similarly, in line 165, TIX MAXLEN is changed to TIXR T
- Immediate addressing mode is used whenever possible.
  - Operand is already included in the fetched instruction. There is no need to fetch the operand from the memory.
  - Denoted by prefix #.

SIC\XE ASSEMBLER…

- Indirect addressing mode is used whenever possible.
  - Just one instruction rather than two is enough.
  - Denoted by the prefix @.
- Instructions referring memory are normally assembled by PC relative or base relative mode.
- If displacements for both PC relative & base relative mode are too large to t into a 3-byte instruction, then 4-byte extended format is used.
  - Denoted by the prefix +.
- Larger main memory of SIC/XE means, has more room to load & run several programs at the same time.
- This kind of sharing of the machine between programs is called multiprogramming.
  - Results in more productive use of hardware.
- To take full advantage of this feature, we must be able to load programs into memory wherever there is room, rather than specifying a fixed address.
  - This introduces the idea of relocation.

INSTRUCTION FORMATS AND ADDRESSING MODES SIC\XE

- Immediate addressing: op#c
- Indirect addressing: op @m
- PC-relative or Base-relative addressing: op m
- The assembler directive BASE is used with base-relative addressing
- If displacements are too large to t into a 3-byte instruction, then 4-byte extended format is used
- Extended format: +op m
- Indexed addressing: op m, x
- Register-to-register instructions
- Large memory
  - Support multiprogramming and need program reallocation capability

A SIC/XE PROGRAM

For relocation

Format 4

```
  5   COPY     START   0         COPY FILE FROM INPUT TO OUTPUT
 10   FIRST    STL     RETADR    SAVE RETURN ADDRESS
 12            LDB     #LENGTH   ESTABLISH BASE REGISTER
 13            BASE    LENGTH
 15   CLOOP   +JSUB    RDREC     READ INPUT RECORD
 20            LDA     LENGTH    TEST FOR EOF (LENGTH = 0)
 25            COMP    #0
 30            JEQ     ENDFIL    EXIT IF EOF FOUND
 35           +JSUB    WRREC     WRITE OUTPUT RECORD
 40            J       CLOOP     LOOP
 45   ENDFIL   LDA     EOF       INSERT END OF FILE MARKER
 50            STA     BUFFER
 55            LDA     #3        SET LENGTH = 3
 60            STA     LENGTH
 65           +JSUB    WRREC     WRITE EOF
 70            J       @RETADR   RETURN TO CALLER
 80   EOF      BYTE    C'EOF'
 95   RETADR   RESW    1
100   LENGTH   RESW    1         LENGTH OF RECORD
105   BUFFER   RESB    4096      4096-BYTE BUFFER AREA
110   .
```

Immediate addressing

Indirect addressing

A SIC/XE PROGRAM…

```
115     .          SUBROUTINE TO READ RECORD INTO BUFFER
120     .
125     RDREC      CLEAR     X              CLEAR LOOP COUNTER
130                CLEAR     A              CLEAR A TO ZERO
132                CLEAR     S              CLEAR S TO ZERO
133               +LDT      #4096
135     RLOOP      TD        INPUT          TEST INPUT DEVICE
140                JEQ       RLOOP          LOOP UNTIL READY
145                RD        INPUT          READ CHARACTER INTO REGISTER A
150                COMPR     A,S            TEST FOR END OF RECORD (X'00')
155                JEQ       EXIT           EXIT LOOP IF EOR
160                STCH      BUFFER,X       STORE CHARACTER IN BUFFER
165                TIXR      T              LOOP UNLESS MAX LENGTH
170                JLT       RLOOP            HAS BEEN REACHED
175     EXIT       STX       LENGTH         SAVE RECORD LENGTH
180                RSUB                     RETURN TO CALLER
185     INPUT      BYTE      X'F1'          CODE FOR INPUT DEVICE
195     .
```

A SIC/XE PROGRAM…

```
        .
        .       SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
        WRREC   CLEAR   X               CLEAR LOOP COUNTER
                LDT     LENGTH
        WLOOP   TD      OUTPUT          TEST OUTPUT DEVICE
                JEQ     WLOOP           LOOP UNTIL READY
                LDCH    BUFFER,X        GET CHARACTER FROM BUFFER
                WD      OUTPUT          WRITE CHARACTER
                TIXR    T               LOOP UNTIL ALL CHARACTERS
                JLT     WLOOP             HAVE BEEN WRITTEN
                RSUB                    RETURN TO CALLER
        OUTPUT  BYTE    X'05'           CODE FOR OUTPUT DEVICE
                END     FIRST
```

SIC/XE Program with Object Code

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	

SIC/XE Program with Object Code…

```
110         .
115         .         SUBROUTINE TO READ RECORD INTO BUFFER
120         .
125   1036  RDREC     CLEAR     X                     B410
130   1038            CLEAR     A                     B400
132   103A            CLEAR     S                     B440
133   103C           +LDT       #4096                 75101000
135   1040  RLOOP     TD        INPUT                 E32019
140   1043            JEQ       RLOOP                 332FFA
145   1046            RD        INPUT                 DB2013
150   1049            COMPR     A,S                   A004
155   104B            JEQ       EXIT                  332008
160   104E            STCH      BUFFER,X              57C003
165   1051            TIXR      T                     B850
170   1053            JLT       RLOOP                 3B2FEA
175   1056  EXIT      STX       LENGTH                134000
180   1059            RSUB                            4F0000
185   105C  INPUT     BYTE      X'F1'                 F1
195
```

SIC/XE Program with Object Code…

```
195                  .
200                  .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205                  .
210     105D    WRREC     CLEAR     X                   B410
212     105F              LDT       LENGTH              774000
215     1062    WLOOP     TD        OUTPUT              E32011
220     1065              JEQ       WLOOP               332FFA
225     1068              LDCH      BUFFER,X            53C003
230     106B              WD        OUTPUT              DF2008
235     106E              TIXR      T                   B850
240     1070              JLT       WLOOP               3B2FEF
245     1073              RSUB                          4F0000
250     1076    OUTPUT    BYTE      X'05'               05
255                       END       FIRST
```

Translation of SIC/XE instructions

- Register translation
  - Register name (A, X, L, B, S, T, F, PC, SW) and their values (0,1, 2, 3, 4, 5, 6, 8, 9).
  - Preloaded in SYMTAB.
- Address translation
  - Most register-memory instructions use program-counter relative or base relative addressing.
  - Format 3: 12-bit address field (disp)
    - base-relative: 0~4095
    - pc-relative: -2048~2047
  - Format 4: 20-bit address field

PC-relative Addressing Mode

- PC-relative or base-relative addressing mode is preferred over direct addressing mode.
  - Save one byte from using format 3 rather than format 4.
  - Reduce program storage space.
  - Reduce program instruction fetch time.

PC-relative Addressing Mode

10	0000	FIRST	STL	RETADR 17202D
12	0003		LDB	#LENGTH

Eg: 1

op (6bits)	n	i	x	b	p	e	disp (12bits)
0001 01	1	1	0	0	1	0	$02D_{16}$

14_{16} + 3_{16}
$=17_{16}$

2_{16}

$$
\begin{aligned}
disp &= TA - (PC) \\
&= RETADR - (PC) \\
&= 0030_{16} - 0003_{16} \\
&= 02D_{16}
\end{aligned}
$$

- After fetching this instruction and before executing it, the PC will be 0003, the address of the next instruction (line 12)
- n & i both set to 1, indicates neither indirect or immediate addressing, which makes 17 instead of 14

PC-relative Addressing Mode…

40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	

Eg: 2

op (6bits)	n	i	x	b	p	e	disp (12bits)
0011 11	1	1	0	0	1	0	FEC_{16}

$3C_{16}$ + 3_{16}

$=3F_{16}$

2_{16}

disp = TA – (PC)

= CLOOP – (PC)

= $0006_{16} - 001A_{16}$

= FEC_{16}

disp could be negative!

PC-relative Immediate Addressing Modes

12	0003		LDB	#LENGTH	69202D
15	0006	CLOOP	+JSUB	RDREC	

Eg: 3

op (6bits)	n	i	x	b	p	e	disp (12bits)
0110 10	0	1	0	0	1	0	$02D_{16}$

68_{16} + 1_{16}
$=69_{16}$

2_{16}

$$\begin{aligned} \text{disp} &= \text{TA} - (\text{PC}) \\ &= \text{LENGTH} - (\text{PC}) \\ &= 0033_{16} - 0006_{16} \\ &= 02D_{16} \end{aligned}$$

- The immediate operand is the value of the symbol LENGTH, which is the address assigned to LENGTH

Base-relative vs. PC-relative

- The assembler knows the value of PC when it tries to use PC-relative mode to assemble an instruction.
- When trying to use base-relative mode to assemble an instruction, the assembler does not know the value of the base register:
  - The programmer must tell the assembler the value of register B.
  - This is done through the use of the BASE directive. Also, the programmer must load the appropriate value into register B by himself.
  - NOBASE can also be used to tell the assembler that no more base-relative addressing mode should be used.

Base-relative Addressing Mode

- BASE register and directive:

  12 LDB #LENGTH

  13 BASE LENGTH

  - Base register is under the control of programmer
  - BASE directive tells assembler that LENGTH is base address;
- BASE : tell the assembler what the base register will contain
- NOBASE : tell the assembler that the contents of the base register can no longer be used for addressing.

Base-relative Addressing Mode…

105	0036	BUFFER	RESB	4096	
160	104E	STCH	BUFFER,X	57C003

Eg: 4

op (6bits)	n	i	x	b	p	e	disp (12bits)
0101 01	1	1	1	1	0	0	003_{16}

$54_{16} + 3_{16}$
$=57_{16}$

C_{16}

$$\begin{aligned} disp &= TA - (B) \\ &= BUFFER - (B) \\ &= 0036_{16} - 0033_{16} \\ &= 003_{16} \end{aligned}$$

Why cannot we use PC-relative?

$$\begin{aligned} disp &= TA - PC = BUFFER - PC \\ &= 0036_{16} - 1051_{16} = EFE5_{16} \text{ (Overflow!)} \end{aligned}$$

Immediate Addressing Mode

55   0020   LDA   #3   010003

Eg: 5

op (6bits)	n	i	x	b	p	e	disp (12bits)
0000 00	0	1	0	0	0	0	003_{16}

00_{16} + 1_{16}
$=01_{16}$

0_{16}

disp = TA
= 003_{16}

Immediate Addressing Mode…

133 103C +LDT #4096 75101000

Eg: 6

op (6bits)	n	i	x	b	p	e	disp (20bits)
0111 01	0	1	0	0	0	1	01000_{16}

74_{16} + 1_{16}

$=75_{16}$

1_{16}

disp = TA

$= 01000_{16}$

USING FORMAT 4

15 0006 CLOOP +JSUB RDREC 4B101036

Eg: 7

op (6bits)	n	i	x	b	p	e	disp (20bits)
0100 10	1	1	0	0	0	1	01036_{16}

48_{16} + 3_{16}

$=4B_{16}$

1_{16}

disp = TA

= 01036_{16}

Indirect Address Mode

- Target addressing is computed as usual (PC-relative or BASE-relative)
- Only the n bit is set to 1

Eg: 8

70	002A	J	@RETADR	3E2003
80	002D	EOF	BYTE C'EOF'	

op (6bits)	n	i	x	b	p	e	disp (12bits)
0011 11	1	0	0	0	1	0	003_{16}

$3C_{16}$ + 2_{16}

$=3E_{16}$

2_{16}

$$\begin{aligned} disp &= TA - (PC) \\ &= RETADR - (PC) \\ &= 0030_{16} - 002D_{16} \\ &= 003_{16} \end{aligned}$$

Program relocation

- Sometimes it is required to load and run several programs at the same time.
- The system must be able to load these programs wherever there is place in the memory.
- Therefore the exact starting is not known until the load time
- The program shown in SIC assembler species that it must be loaded at address 1000 for correct proper execution. This restriction is too inflexible for the loader.
- If the program is loaded at a different address, say 2000, its memory references must be modified. For example,

55 101B LDA THREE 00102D

55 101B LDA THREE 00202D

- Thus, we wish programs relocatable so that they can be loaded and execute correctly at any place in the memory.

CON…

- Only those instructions that use absolute (direct) addresses to reference symbols.
- The following need not be modified
  - Immediate addressing (no memory references)
  - PC or Base-relative addressing (Relocatable is one advantage of relative addressing).
  - Register-to-register instructions (no memory references).

Machine independent assembler features

- Literals
- Symbol-Defining Statements
- Expressions
- Program Blocks
- Control Sections

Literals

- Let programmers to be able to write the value of a constant operand as a part of the instruction that uses it.
- This avoids having to define the constant elsewhere in the program and make up a label for it.
- Such an operand is called a literal because the value is stated “literally" in the instruction.
- A literal is defined with a prefix = followed by a specification of the literal value.

Examples:

45 001A ENDFIL LDA =C'EOF' 032010

215 1062 WLOOP TD =X'05' E32011

Literals…

Data Structure: literal table - LITTAB

Content:

- Literal name
- The operand value and length
- Address assigned to the operand

Implementation:

- Organized as a hash table, using literal name or value as key.

How the Assembler Handles Literals?

Pass 1:

- Build LITTAB with literal name, operand value and length, (leaving the address unassigned).

Pass 2:

- Search LITTAB for each literal operand encountered

Symbol-Defining Statements

- Users can define labels on instructions or data areas
- The value of a label is the address assigned to the statement
- Users can also define symbols with values symbol EQU value
- EQU: Assembler directive
- value can be: constant, other symbol, expression
- It uses to make the source program easier to understand(advantage of SDS)
- No forward reference

Symbol-Defining Statements

How assembler handles it?

- In pass 1: when the assembler encounters the EQU statement, it enters the symbol into SYMTAB for later reference.
- In pass 2: assemble the instruction with the value of the symbol Follow the previous approach

Example 1: assign mnemonic code for registers

A EQU 0

X EQU 1

L EQU 2

Example 2:

MAXLEN EQU 4096

+LDT #MAXLEN

EXPRESSIONS

- Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /.
- The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location.
- Thus the statement

BUFFEND EQU *

assigns a value to BUFFEND, which is the address of the next byte following the buffer area.

Program blocks:

- Collect many pieces of code/data that scatter in the source program but have the same kind into a single block in the generated object program
- In a different order by Separating blocks for storing code, data, stack, and larger data block.
- Example : three blocks are used:

Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory

Advantage:

- Because pieces of code are closer to each other now, format 4 can be replaced with format 3, saving space and execution time
- Data protection can better be done

Program-Block Example

There is a default block

```
COPY     START   0              COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR         SAVE RETURN ADDRESS
CLOOP    JSUB    RDREC          READ INPUT RECORD
         LDA     LENGTH         TEST FOR EOF (LENGTH = 0)
         COMP    #0
         JEQ     ENDFIL         EXIT IF EOF FOUND
         JSUB    WRREC          WRITE OUTPUT RECORD
         J       CLOOP          LOOP
ENDFIL   LDA     =C'EOF'        INSERT END OF FILE MARKER
         STA     BUFFER
         LDA     #3             SET LENGTH = 3
         STA     LENGTH
         JSUB    WRREC          WRITE EOF
         J       @RETADR        RETURN TO CALLER
         USE     CDATA
RETADR   RESW    1
LENGTH   RESW    1              LENGTH OF RECORD
         USE     CBLKS
BUFFER   RESB    4096           4096-BYTE BUFFER AREA
BUFEND   EQU     *              FIRST LOCATION AFTER BUFFER
MAXLEN   EQU     BUFEND-BUFFER  MAXIMUM RECORD LENGTH
```

The CBLKS block contain all data areas that consist of large blocks of

Control Sections

- A part of the program that maintains its identity after assembly
- Can be loaded and relocated independently of the others
- The programmer can assemble, load, and manipulate each of these control sections separately
- Instructions in one control section may need to refer to instructions or data located in another control section
- The references that are between control sections are called "external references
- EXTREF names symbols that are used in this control section

Control-Section Example…

A new control section

```
          WRREC     CSECT
          .
          .         SUBROUTINE TO WRITE RECORD FROM BUFFER
          .
                    EXTREF    LENGTH,BUFFER
                    CLEAR     X                 CLEAR LOOP COUNTER
                   +LDT       LENGTH
          WLOOP     TD        =X'05'            TEST OUTPUT DEVICE
                    JEQ       WLOOP             LOOP UNTIL READY
                   +LDCH      BUFFER,X          GET CHARACTER FROM BUFFER
                    WD        =X'05'            WRITE CHARACTER
                    TIXR      T                 LOOP UNTIL ALL CHARACTERS
                    JLT       WLOOP               HAVE BEEN WRITTEN
                    RSUB                        RETURN TO CALLER
                    END       FIRST
```

Assembler design options

- Design is done in two passes. One pass and multi pass assembler

One pass assembler :

- Main problem in designing the assembler using single pass was to resolve forward references(data items, labels on instructions

Solution:

- data items: require a programmer to define variables before using them
- labels on instructions:
  - sometimes the program needs a forward jump
  - to use only backward jumps is too restrictive
  - no good solution

SIC PROGRAM FOR ONE PASS ASSEMBLER

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110		.			

MULTI-PASS ASSEMBLER

- So a multi-pass assembler resolves the forward references and then converts into the object code.
- Hence the process of the multi-pass assembler can be as follows:
- *Pass-1*
  - Assign addresses to all the statements
  - Save the addresses assigned to all labels to be used in *Pass-2*
  - Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
  - Defines the symbols in the symbol table(generate the symbol table)

Con…

- *Pass-2*
  - Assemble the instructions (translating operation codes and looking up addresses).
  - Generate data values defined by BYTE, WORD etc.
  - Perform the processing of the assembler directives not done during *pass-1*.
  - Write the object program and assembler listing.
- The most important things which need to be concentrated is the generation of *Symbol table* and resolving *forward references*.

CON…

- 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.

Multi-Pass Assembler Implementation

- Use a symbol table to store symbols that are not totally defined yet
- For an undefined symbol
  - Store the names and the number of undefined symbols which contribute to the calculation of its value
  - Keep a list of symbols whose values depend on the defined value of this symbol
  - When the symbol becomes defined, use its value to reevaluate the values of all of the symbols that are kept in this list
- Perform the above steps recursively

Thank you

Question?