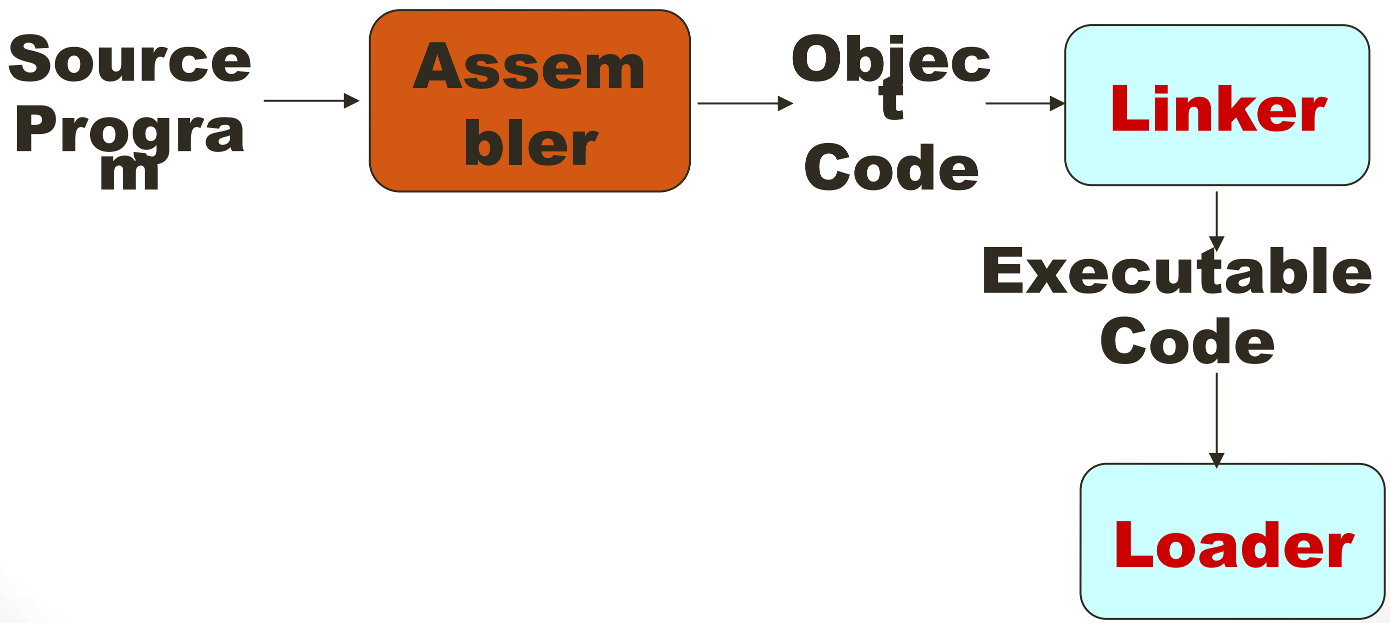
DAYANANDA SAGAR UNIVERSITY

- DEPARTMENT OF CSE

Chapter 2
Assemblers



Outline

- 2.1 Basic Assembler Functions
  - A simple SIC assembler
  - Assembler tables and logic
- 2.2 Machine-Dependent Assembler Features
  - Instruction formats and addressing modes
  - Program relocation
- 2.3 Machine-Independent Assembler Features
- 2.4 Assembler Design Options
  - Two-pass
  - One-pass
  - Multi-pass

Why an Assembly Language is needed?

- Programming in machine code, by supplying the computer with the numbers of the operations it must perform, can be quite a burden, because for every operation the corresponding number must be looked up or remembered.
- Looking up all numbers takes a lot of time, and mis-remembering a number may introduce computer bugs.
- So Assembly Languages are evolved which contains mnemonic instructions corresponding to the Machine codes using which the program can be written easily.
- Therefore a set of mnemonics was devised. Each number was represented by an alphabetic code. So instead of entering the number corresponding to addition to add two numbers one can enter "add".
- Although mnemonics differ between different CPU designs some are common, for instance: "sub" (subtract), "div" (divide), "add" (add) and "mul" (multiply).

What is an Assembler?

- An assembler is system software which is used to convert an assembly language program to its equivalent object code. The input to an assembler is a source code written in assembly language (using mnemonics) and the output is the object code.

Format of Assembly language program

1.Label field.

The label is a symbolic name that represents the memory address of an executable statement or a variable.

2.Opcode/directive fields.

The opcode (e.g. operation code) specifies the symbolic name for a machine instruction.

The directive specifies commands to the assembler about the way to assemble the program.

3.**Operand field.**

The operand specifies the data that is needed by a statement.

4.Comment field.

The comment provides clear explanation for a statement.

Basic Assembler Functions

The basic assembler functions are

- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.

- Convert *mnemonic operation codes* to their *machine language equivalents*
  E.g. STL -> 14 (line 10)
- Convert *symbolic operands* to their equivalent *machine addresses*
  E.g. RETADR -> 1033 (line 10)
- Build the machine instructions in the proper format
- Convert the *data constants* to *internal machine representations*
  E.g. EOF -> 454F46 (line 80)
- Write the *object program* and the *assembly listing*

Assembler Directives

Assembler directives are Pseudo-instructions that are not translated into machine instructions and they provide instructions to the assembler itself.

- **The SIC assembler directives.**
  - START
    - Specification of the name and start address of the program.
  - END
    - Indication of the end of the program and optionally the address of the first executable instruction.
  - BYTE
    - Generate character or hexadecimal constant occupying as many as needed to represent the constant.
  - WORD
    - Generate one word constant.
  - RESB
    - Reserve the indicated number of bytes for a data area.
  - RESW
    - Reserve the indicated number of words for a data area.

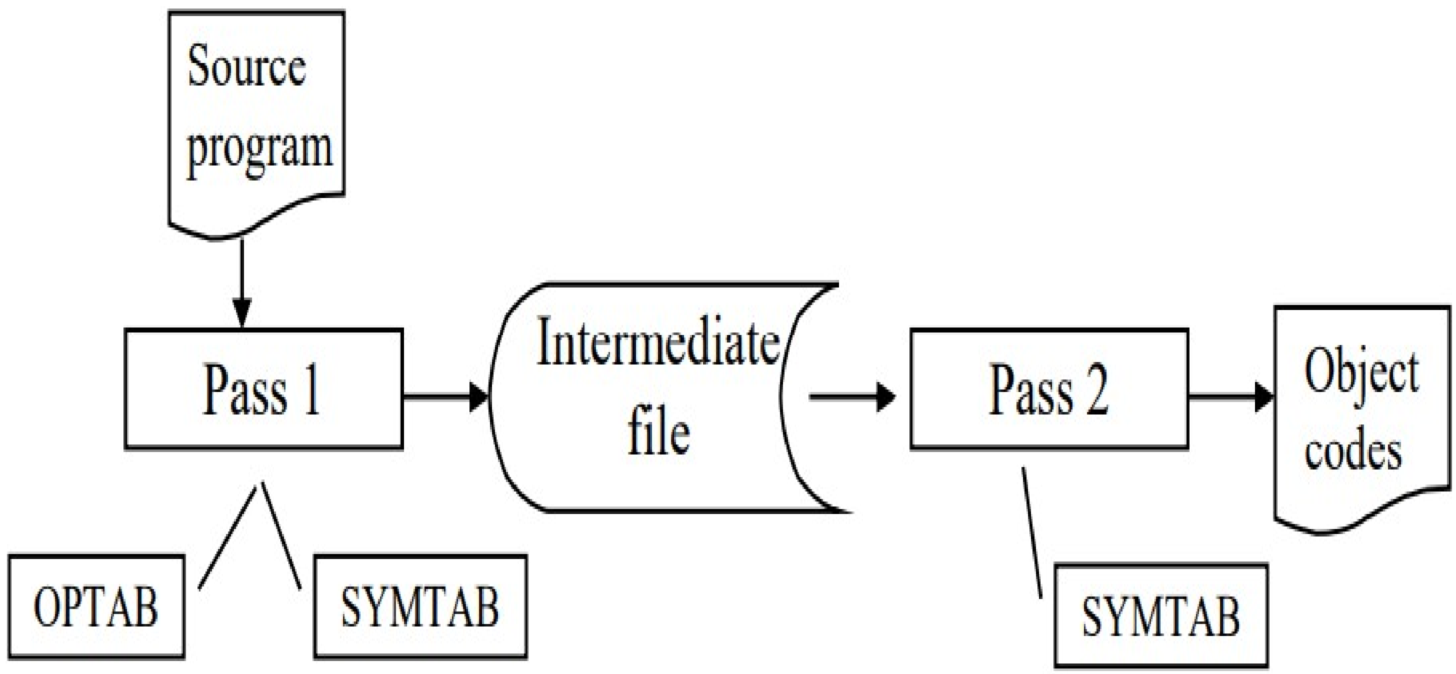
Functions of Two pass Assembler

Functions of Two Pass Assembler

- **Pass 1 - Define symbols (assign addresses)**
  - Assign addresses to all statements in the program
  - Save the values assigned to all labels for use in Pass 2
  - Process some assembler directives
- **Pass 2 - Assemble instructions and generate object program**

  Assemble instructions
  - Generate data values defined by BYTE, WORD, etc.
  - Process the assembler directives not done in Pass 1
  - Write the object program and the assembly listing

Functions of Two Pass Assembler



Object Code Generation for SIC Assembler Language Program

Note:

- Add 3 to LOCCTR (Instruction 3 Format)
- Add 4 to LOCCTR(Instruction 4 Format)(Instructions prefixed with + sign)

- OPCODE =WORD→Add **3** to LOCCTR

- OPCODE =RESW-→Add **3 * Value of Operand** to LOCCTR

- OPCODE =BYTE-→Add **length of constant in bytes** to LOCCTR

- OPCODE =RESB-→Add **Value of operand** to LOCCTR

Example 1: SIC program
Given Opcode values and Hex Codes

Mnemonic	Opcode
STL	14
JSUB	48
COMP	28
JEQ	30
J	3C
LDA	00
STA	0C

ASCII CODE	HEX Code
E	45
O	4F
F	46

LOCCTR	Labels	Opcode	operand	Object Code
	COPY	START	1000	
1000	FIRST	STL	RETADR	141024
1003	CLOOP	JSUB	RDREC	482039
1006		LDA	LENGTH	001021
1009		COMP	ZERO	28101E
100C		JEQ	ENDFIL	301015
100F		JSUB	WRREC	482061
1012		J	CLOOP	3C1003
1015	ENDFIL	LDA	EOF	00101B
1018		STA	LENGTH	OC1021
101B	EOF	BYTE	C'EOF'	454F46
101E	ZERO	WORD	0	000000
1021	LENGTH	RESW	1	
1024	RETADR	RESW	1	
1027	. .	. .	. .	. .
2039	RDREC	LDX	ZERO	
	….			
2061	WRREC	LDX	ZERO	

Example 2: SIC Program
Given Opcode values and Hex Codes

Mnemonic	Opcode
LDX	04
LDA	00
TD	E0
JEQ	30
RD	D8
COMP	28
STCH	54
TIX	2C
JLT	38
STX	10
RSUB	4C

ASCII CODE	HEX Code
T	54
E	45
S	53

LOCCTR	Labels	Opcode	operand	Object Code
	COPY	START	2039	
2039	RDREC	LDX	CLOOP	04204E
203C		LDA	CLOOP	00204E
203F	RLOOP	TD	INPUT	E0205D
2042		JEQ	RLOOP	30203F
2045		RD	INPUT	D8205D
2048		COMP	EXIT	282057
204B		JEQ	RDREC	302039
204E	CLOOP	STCH	EXIT	542057
2051		TIX	MAXLEN	2C205E
2054		JLT	RLOOP	38203F
2057	EXIT	STX	EOF	103061
205A		RSUB		4C0000
205D	INPUT	BYTE	X 'F1'	F1
205E	MAXLEN	WORD	0	000000
2061	BUFFER	RESB	4096	
3061	EOF	BYTE	C 'TEST'	54455354
3065		END	COPY	

Object Program

- **Three types of records**
  - Header: program name, starting address, length.
  - Text: starting address, length, object code.
  - End: address of first executable instruction

Header record:

Col. 1	H
Col. 2–7	Program name
Col. 8–13	Starting address of object program (hexadecimal)
Col. 14–19	Length of object program in bytes (hexadecimal)

Object Program

Text record:

Col. 1	T
Col. 2–7	Starting address for object code in this record(hexadecimal)
Col. 8–9	Length of object code in this record in bytes (hexadecimal)
Col. 10–69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

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

Assembler Data Structures

- Our simple assembler uses two internal tables: The OPTAB and SYMTAB.
  - OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalents.
    - LDA→00, STL→14, …
  - SYMTAB is used to store values (addresses) assigned to labels.
    - FIRST→1000, COPY→1000, …

- Location Counter LOCCTR
  - LOCCTR is a variable for assignment addresses.
  - LOCCTR is initialized to address specified in START.
  - When reach a label, the current value of LOCCTR gives the address to be associated with that label.

The Operation Code Table (OPTAB)

- Contain the mnemonic operation & its machine language equivalents (at least).
- Contain instruction format & length.
- Pass 1, OPTAB is used to look up and validate operation codes.
- Pass 2, OPTAB is used to translate the operation codes to machine language.
- In SIC/XE, assembler search OPTAB in Pass 1 to find the instruction length for incrementing LOCCTR.
- Organize as a hash table (static table).

mnemonic	Opcode	format
STL	14	3
LDB	68	3
JSUB	48	4
LDA	00	3

The Symbol Table (SYMTAB)

- Include the name and value (address) for each label.
- Include flags to indicate error conditions
- Contain type, length.
- Pass 1, labels are entered into SYMTAB, along with assigned addresses (from LOCCTR).
- Pass 2, symbols used as operands are look up in SYMTAB to obtain the addresses.
- Organize as a hash table (static table).
- The entries are rarely deleted from table.

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

PASS 1 of Two Pass Assembler

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
```

PASS 1 of Two Pass Assembler

```
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
                        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
                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}
```

PASS 2 of Two Pass Assembler

```
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
```

PASS 2 of Two Pass Assembler

```
while OPCODE ≠ 'END' do
   begin
      if this is not a comment line then
         begin
            search OPTAB for OPCODE
            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
            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}
```

Object Code Generation-SIC/XE

Note:

- Add 3 to LOCCTR (Instruction 3 Format)
- Add 4 to LOCCTR(Instruction 4 Format)(Instructions prefixed with + sign)

- OPCODE =WORD→Add **3** to LOCCTR
- OPCODE =RESW→Add **3 * Value of Operand** to LOCCTR
- OPCODE =BYTE-→Add **length of constant in bytes** to LOCCTR
- OPCODE =RESB-→Add **Value of operand** to LOCCTR

- Set n= 1 for opcode @m (indirect)
- Set i=1 for opcode #c (immediate)
- Set n=i=1 for simple addressing mode
- Set x=1 for opcode m,x (indexed)
- Set e=1 for instruction format 4

Object Code Generation-SIC/XE

LOCCTR	Label	Opcode	Operand	Object Code
	COPY	START	0	
	FIRST	STL	RETADR	
		LDB	#LENGTH	
	CLOOP	+JSUB	RDREC	
		LDA	LENGTH	
		COMP	#0	
		JEQ	ENDFIL	
		J	CLOOP	
	ENDFIL	LDA	EOF	
	RDREC	+LDT	#4096	
	EOF	BYTE	C ‘EOF’	
	RETADR	RESW	1	
	LENGTH	RESW	1	
		END	COPY	

Example 1:
Given Opcode values and Hex Codes

Mnemonic	Opcode
STL	14
LDB	68
JSUB	48
LDA	00
COMP	28
JEQ	30
J	3C
LDT	74

ASCII CODE	HEX Code
E	45
O	4F
F	46

Object Code Generation-SIC/XE

LOCCTR	Label	Opcode	Operand	Object Code
	COPY	START	0	
0000	FIRST	STL	RETADR	
0003		LDB	#LENGTH	
0006	CLOOP	+JSUB	RDREC	
000A		LDA	LENGTH	
000D		COMP	#0	
0010		JEQ	ENDFIL	
0013		J	CLOOP	
0016	ENDFIL	LDA	EOF	
0019	RDREC	+LDT	#4096	
001D	EOF	BYTE	C ‘EOF’	
0020	RETADR	RESW	1	
0023	LENGTH	RESW	1	
0026		END	COPY	

Object Code Generation-SIC/XE

- **FIRST STL RETADR**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	disp(12)
000000	1	1	0	0	1	0	**017**

- **STL→opcode→14**

Add (STL value)+ (op,n,i)→14+3→17

- **x,b,p,e→ 0010→2**

- **Disp=TA-(PC)**
  - **=(RETADR Address)-(PC)**
  - **=0020-003**
  - **=017**

Object code is 172017

Object Code Generation-SIC/XE

- **LDB #LENGTH**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	disp(12)
000000	0	1	0	0	1	0	**01D**

- **LDB→ opcode→68**

Add LDB Value+ (op,n,i)= 68+ 1 →69

- **x,b,p,e→ 0010→2**

- **Disp= TA-(PC)**

  =(LENGTH Address) – (0006)

  =(0023)-(0006)

  =001D

Object Code==69201D

Object Code Generation-SIC/XE

- CLOOP +JSUB RDREC

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Address(20 bits)
000000	1	1	0	0	0	1	00019

- **JSUB→opcode→48**

Add (JSUB value)+ (op,n,i)→48+3→4B

Instruction 4 format -> Set e=1

- **x,b,p,e→ 0001→1**

- **Address= 0019**

Object code is 4B100019

Object Code Generation-SIC/XE

- **LDA LENGTH**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Disp(12bits)
000000	1	1	0	0	1	0	**016**

- **LDA→opcode→00**

Add (LDA value)+ (op,n,i)→00+3→03

- **x,b,p,e→ 0010→2**

- **Disp=(TA)-(PC)**
  - **=(LENGTH address)-(000D)**
  - **=0023-000D**
  - **=0016**

Object code is 032016

Object Code Generation-SIC/XE

- **COMP #0**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Disp(12bits)
000000	0	1	0	0	0	0	**000**

- **COMP→opcode→28**

Add (COMP value)+ (op,n,i)→28+1→29

- **x,b,p,e→ 0000→0**

- **Disp=000**

Object code is 290000

Object Code Generation-SIC/XE

- **JEQ ENDFIL**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Disp(12bits)
000000	1	1	0	0	1	0	**003**

- **JEQ→opcode→30**

Add (JEQ value)+ (op,n,i)→30+3→33

- **x,b,p,e-→ 0010→2**

- **Disp=(TA)-(PC)**
  - **=(ENDFIL address)-(0013)**
  - **=0016-0013**
  - **=0003**

Object code is 332003

Object Code Generation-SIC/XE

- **J CLOOP**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Disp(12bits)
000000	1	1	0	0	1	0	**FF0**

- **J→opcode→3C**

Add (J value)+ (op,n,i)→3C+3-→3F

- **x,b,p,e→ 0010→2**

- **Disp=(TA)-(PC)**
- **=(CLOOP address)-(0016)**
- **=(=0006-0016**
- **== -10**

Convert the 2s complement of -10

10 into binary→ 0001 0000

Convert 0’s to 1’s and 1’s to 0’s→ 1110 1111

Add 1 → 0000 0001

1111 0000

F 0

Value is F0 . To fit into 12 bit displacement field

Value is FF0

Therefore Object code is 3F2FF0

Object Code Generation-SIC/XE

- **ENDFIL LDA EOF**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Disp(12bits)
000000	1	1	0	0	1	0	**004**

- **LDA-→opcode-→00**

Add (LDA value)+ (op,n,i)→00 + 3 →03

- **x,b,p,e-→ 0010-→2**

- **Disp=(TA)-(PC)**
- **=(EOF address)-(0019)**
- **=(=001D-0019)**
- **=004**

Object Code = 032004

Object Code Generation-SIC/XE

- **RDREC +LDT #4096**

Op(6)	n(1)	i(1)	x(1)	b(1)	p(1)	e(1)	Addr(20bits)
000000	0	1	0	0	0	1	**01000**

- **LDT→opcode→74**

Add (LDT value)+ (op,n,i)→74 + 1 →75

- **x,b,p,e→0001→1**

Address= Immediate value = 4096

Convert 4096 to hex equivalent = 1000h

Object Code = 75101000

Problem 2: Object Code Generation-SIC/XE

LOCCTR	Label	Opcode	Operand	Object Code
	SUM	START	0	
	FIRST	CLEAR	X	
		LDA	#0	
		+LDB	#TOTAL	
		BASE	TOTAL	
	LOOP	ADD	TABLE,X	
		TIX	COUNT	
		JLT	LOOP	
		+STA	TOTAL	
	COUNT	RESW	1	
	TABLE	RESW	2000	
	TOTAL	RESW	1	
		END	FIRST	

Object Code Generation-SIC/XE

LOCCTR	Label	Opcode	Operand	Object Code
	SUM	START	0	
0000	FIRST	CLEAR	X	B410
0002		LDA	#0	010000
0005		+LDB	#TOTAL	69101789
		BASE	TOTAL	
0009	LOOP	ADD	TABLE,X	1BA00D
000C		TIX	COUNT	2F2007
000F		JLT	LOOP	382FF7
0012		+STA	TOTAL	0F101789
0016	COUNT	RESW	1	
0019	TABLE	RESW	2000	
1789	TOTAL	RESW	1	
178C		END	FIRST	

Machine dependent features of a SIC/XE Assembler

1.Instruction formats

2. Addressing modes

3. Program relocation

Instruction Formats

- The instruction formats depend on **the memory organization and the size of the memory.**
- In **SIC machine** the memory is byte addressable. Word size is 3 bytes. The size of the memory is 32768(2^15) bytes
- Accordingly it supports only one instruction format. It has only two registers: register A and Index register. Therefore the addressing modes supported by this architecture are direct, indirect, and indexed.
- Whereas the memory of a **SIC/XE** machine is 1MegaByte(2^20) bytes.
- This supports four different types of instruction types, they are

a. 1 byte instruction

b.2 bytes instruction

c. 3 bytes instruction

d. 4 bytes instruction

Instruction formats

Format 1 (1 byte)		
op(8)		
Format 2 (2 bytes)		
op(8)	r1(4)	r2(4)

Format 3(3 bytes)

op(6)	n	i	x	b	p	e	disp(12)
	1	1	1	1	1	1	

Format 4(4 bytes)

op(6)	n	i	x	b	p	e	address (20)
	1	1	1	1	1	1	

Addressing Modes

Addressing Modes are:

– Index Addressing(SIC): Opcode m, x

– Indirect Addressing: Opcode @m

– PC-relative: Opcode m

– Base relative: Opcode m

– Immediate addressing: Opcode #c

Program Relocation

- **Principles.**
- o The load address of an object program is unknown at assembly time if the system implements the multiprogramming feature.
- o The assembler generates addresses relative to zero in the object program.
- o At load time, relocation is performed by adding the load address to the relative addresses.
- o Operands of instructions that use direct addressing must be relocated, and the assembler provides the relocation information in the object program.
- o Operands of instructions that use relative addressing do not need to be relocated.
- o Relocation can be processed by the loader or by the CPU using relocation registers

Program Relocation

- Absolute Program

In this the address is mentioned during assembling itself. This is called Absolute Assembly.

- Consider the instruction:

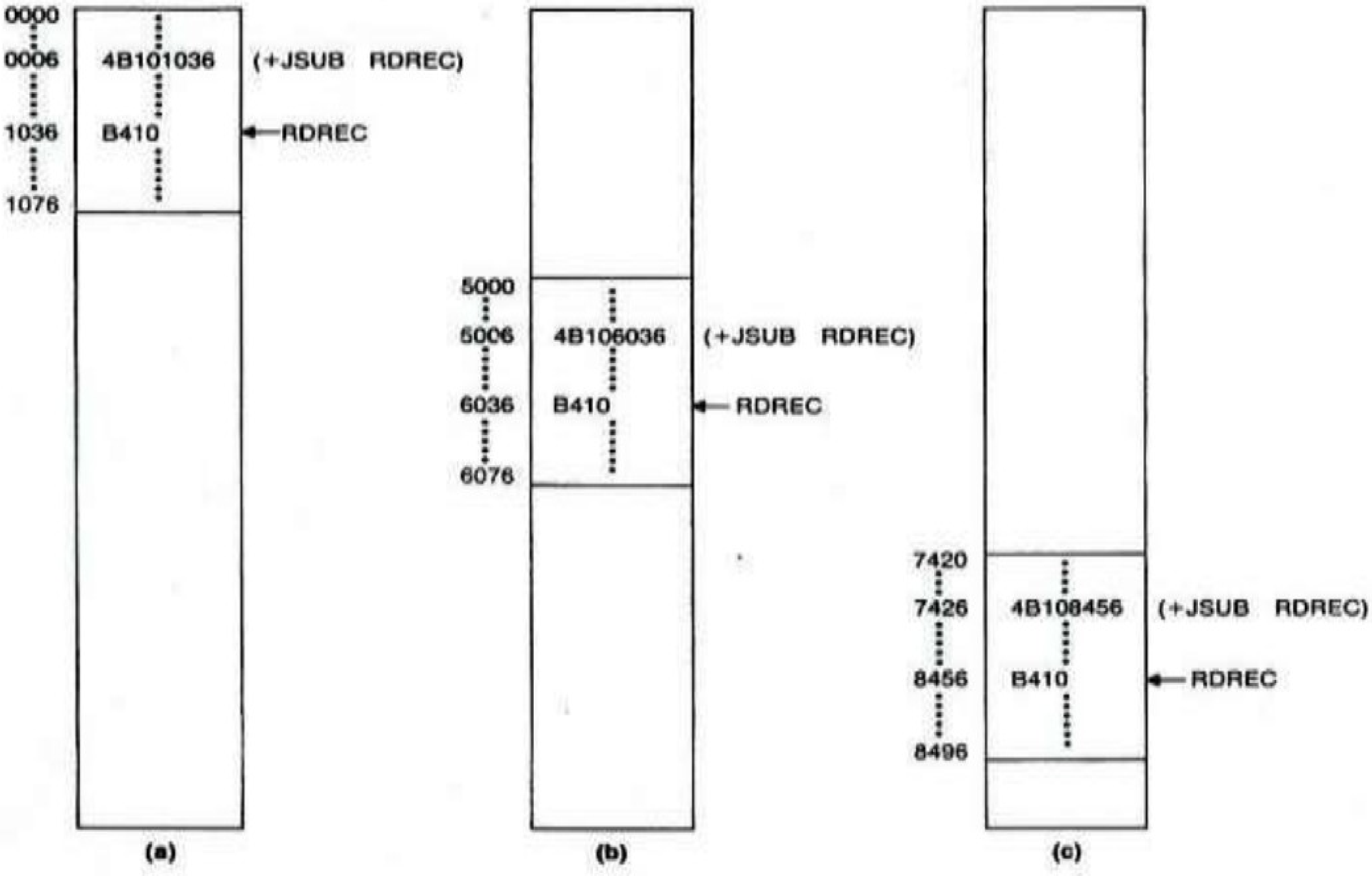
55 101B LDA THREE 00102D

- This statement says that the register A is loaded with the value stored at location 102D.Suppose it is decided to load and execute the program at location 2000 instead of location 1000. Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.

Program Relocation

- Apart from the instruction which will undergo a change in their operand address value as the program load address changes.
- There exist some parts in the program which will remain same regardless of where the program is being loaded.
- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program.
- However, the assembler identifies for the loader those parts of the program which need modification. An object program that has the information necessary to perform this kind of modification is called the **relocatable program.**

Program Relocation



Program Relocation

- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC
- . The second figure shows that if the program is to be loaded at new location 5000. The address of the instruction JSUB gets modified to new location 6036.
- Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- When assembler generates the object code for the JSUB instruction we are considering, it will insert the address of RDREC relative to the start of the program.
- The assembler will produce a command for the loader instructing it to add the beginning address of the program to the address field in the JSUB instruction at load time.
- **From the object program, it is not possible to distinguish the address and constant The assembler must keep some information to tell the loader. The object program that contains the modification record is called a relocatable program.**

Advantages of Program Relocation

1.The larger main memory of SIC/XE

- Several programs can be loaded and run at the same time.
- This kind of sharing of the machine between programs is called multiprogramming

2. To take full advantage

- Load programs into memory wherever there is space.
- Not specifying a fixed address at assembly time.

Modification Record

- Col 1: M
- Col 2-7 Starting location of the address field to be modified relative to the beginning of the program(hexadecimal)
- Col 8-9 length of the address field to be modified, in half bytes(hexadecimal).

Machine Independent Assembler Features

1.Literals

2.Symbol Defining Statements

3.Expressions

4.Program Blocks

5. Control Sections and program linking

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 literally stated in the instruction. Note that a literal is identified with the prefix = which followed by a specification of the literal value.
- The literal is a special type of relocatable term. It behaves like a symbol in that it represents data. However, it is a special kind of term because it also is used to define the constant specified by the literal.

Literals

- This is convenient because:

a. The data you enter as numbers for computation, addresses, or messages to be printed is visible in the instruction in which the literal appears.

b. You avoid the added effort of defining constants elsewhere in your source module and then using their symbolic names in machine instruction operands.

Examples:

Ex 1:

45 001A ENDFIL BYTE =C ''EOF'' 454F46

Specifies a 3 byte operand with value "EOF"

Ex 2:

69101A CLOOP STA = X '05' E32011

Specifies a 1 byte literal with hexadecimal value 05

Literals vs Immediate Operands

- *Immediate Operands*

The operand value is assembled as part of the machine instruction

e.g. 55 0020 LDA #3 010003

- *Literals*

The assembler generates the specified value as a constant at some other memory location

e.g. 45 001A ENDFIL LDA = C'EOF' 032010

Literal pools

- All of the literal operands used in a program are gathered together into one or more literal pools.
- Normally literals are placed into a pool at the end of the program .
- In some cases, it is desirable to place literals into a pool at some other location in the object program .
- For this purpose assembler directive **LTORG is used.**

Reason: Keep the literal operand close to the instruction.

1.When the assembler encounters a LTORG statement , it creates a literal pool that contains all of the literal operands used since the previous LTORG.

2. This literal pool is placed in the object program at the location where the LTORG directive was encountered.

3. Literal placed in a pool by LTORG will not be repeated in the pool at the end of the program

Duplicate literals:

e.g. 215 1062 WLOOP TD =X''05''

e.g. 230 106B WD =X"05''

- The assemblers should recognize duplicate literals and store only one copy of the specified data value .
- Only one data area with this value is generated. Both the instruction refer to the same address in the literal pool for their operand.
- The easiest method to recognize duplicate literals is by:

1.Comparison of the defining expression

Same literal name with different value, e.g. LOCCTR=*

2. Comparison of the generated data value

The benefits of using generate data value are usually not great enough to justify the additional complexity in the assembler .

LITTAB:

- The basic data structure that assembler handles literal operands is literal table. For each literal used, this table contains **the literal name, the operand value, the length and the address assigned to the operan**d when its placed in the literal pool.

Pass 1

a. Build LITTAB with literal name, operand value and length, leaving the address unassigned .

b. When LTORG statement is encountered, assign an address to each literal not yet assigned an address

Pass 2

a. Search LITTAB for each literal operand encountered .

b. Generate the data values using BYTE or WORD statement.

c. Generate modification record for literals that represent an address in the program

Literals

Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
13		LDB	#LENGTH	ESTABLISH BASE REGISTER
14		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	=C'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
93		LTORG		
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
106	BUFEND	EQU	*	
107	MAXLEN	EQU	BUFEND-BUFFER	MAXIMUM RECORD LENGTH

Literals

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
13	0003		LDB	#LENGTH	69202D
14			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	=C'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
93			LTORG		
	002D	*	=C'EOF'		454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
106	1036	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	

Literals

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

Figure 2.10 Program from Fig. 2.9 with object code.

LITTAB

Literal Name	Operand Value	Length	Address
C='EOF'	454F46	3	002D
X='05'	05	1	1076

Symbol defining statements

1. **EQU Directive** **2.ORG Directive**

1.EQU: The user defined symbols in the assembler language program appears as labels on instructions or data areas. The value of such a label is the address assigned to the statement on which it appears. Most assemblers provide an assembler directive that allows the user to define the symbol and specify their values. **EQU is the assembler directive used.**

The general form of such a statement is

Symbol EQU Value

It defines the given symbol and assigns the specified value.

The value may be a

a. Constant

b. Any expression involving constant.

c. Previously defined symbol

EQU Directive

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

1. In the last program, the statement used as

   +LDT   #4096 // to load the value 4096 into register T.

2. If we include the statement

   MAXLEN   EQU   4096

In the program then we can write it as

   +LDT   #MAXLEN

3.When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB with the value 4096.

EQU Directive

4. Another common use of the EQU is defining mnemonic names for registers.

For ex

- A EQU 0
- X EQU 1
- L EQU 2

These statement cause the symbols A,X,L… to be entered into SYMBOL with their corresponding values 0.1,2…

ORG DIRECTIVE:

1. Indirectly assigns values to symbols. This is called Origin directive.

2.Resets the location counter value to the specified value.

3.Since the values of the symbols are taken from the LOCCTR, the ORG statement will affect the values of all the labels defined until the next ORG.

EX:

- To define a symbol table with the following structure:

a. SYMBOL field: 6 bytes

b. VALUE field: one word.

c. FLAGS field : 2 byte

ORG Directive

STAB
(100 entries)

SYMBOL	VALUE	FLAGS

Symbol field contains user defined symbol: Value field represents the value assigned to the symbol and the Flags field specifies symbol type and other information.

To reserve the space we write

STAB RESB 1100

We also define the labels as SYMBOL,VALUE, FLAGSS using with EQU statements:

```
SYMBOL   EQU  STAB
VALUE    EQU  STAB+6
FLAGS    EQU  STAB+9
```

ORG Directive

- To fetch the value field

  LDA VALUE,X

- We can accomplish the same symbol definition using ORG in the following way

```
STAB      RESB    1100
ORG       STAB
SYMBOL    RESB    6
VALUE     RESW    1
FLAGS     RESB    2
          ORG     STAB+1100
```

a. The first ORG resets the LOCCTR to the value of STAB.
b. RESB statement defines the SYMBOL to have the current value in LOCCTR.
c. LOCCTR is then advanced so that the label on the RESW statement assigns to VALUE the address(STAB+6) and so on..
d. So that each entry in STAB consists of 6 byte SYMBOL, followed by one word VALUE, followed by a 2 byte FLAGS.
e. The last ORG statement set LOCCTR back to its previous value.

ORG Directive

Notice that the two pass assembler design requires that all symbols be defined during Pass1.

For ex:

The sequence

ALPHA	RESW	1
BETA	EQU	ALPHA

would be allowed, whereas the sequence

BETA	EQU	ALPHA
ALPHA	RESW	1

would not be allowed.

Reason for this is symbol definition process.

ORG Directive

ORG has the same type of restriction.
Consider the following example

```
ORG          ALPHA
BYTE1        RESB  1
BYTE2        RESB  1
BYTE3        RESB  1
             ORG
ALPHA        RESB  1
```

The above sequence cannot be processed because assembler directive would not know the value assigned to the location counter in response to the first ORG statement.

The symbols BYTE1, BYTE2, BYTE3 cant be assigned addresses during PASS1.

Expressions

- Most assemblers allow the use of the expressions whenever a single operand is permitted.
- Each such expression has to be evaluated by the assembler to produce a single operand address or value.
- Expressions can be classified as

1.ABSOLUTE EXPRESSIONS

2. RELATIVE EXPRESSIONS

Expressions

- **RELATIVE EXPRESSIONS:**
- Relative means relative to the beginning of the program. labels on the instructions and data areas and references to the location counter values are relative terms. No relative term into multiplication or division operation.
- **ABSOLUTE EXPRESSIONS:**
- Absolute means independent of program location. A constant is an absolute term. Absolute expressions may also contain relative terms provided they occur in pair and the terms in each pair have opposite sign.
- A relative term or expression represents some value that may be written as (S+r) where
- S= Starting address of the program
- R= value of the term or expression relative to the starting address

Expressions

Example: 107 MAXLEN EQU BUFFEND-BUFFER

Both BUFFEND and BUFFER are relative terms, each representing an address within the program. However the expression BUFFEND-BUFFER represents the absolute term.

To determine the type of an expression we must keep track of the type of the symbols defined in the program.

Following table shows the symbol table entries.

Symbol	Type	Value
RETADR	R	0000
BUFFER	R	0036
BUFFEND	R	1036
MAXLEN	A	1000

SIC program

LOCCTR	Labels	Opcode	operand	Object Code
	COPY	START	1000	
1000	FIRST	STL	RETADR	141021
1003	CLOOP	JSUB	RDREC	482039
1006		LDA	LENGTH	00101E
1009		COMP	RETADR	281021
100C		JEQ	ENDFIL	301015
100F		JSUB	WRREC	482061
1012		J	CLOOP	3C1003
1015	ENDFIL	LDA	EOF	00101B
1018		STA	LENGTH	OC101E
101B	EOF	BYTE	C’EOF’	454F46
101E	LENGTH	RESW	1	
1021	RETADR	RESW	1	
1024		END	COPY	. .
2039	RDREC			
2061	WRREC			

Object Program

Header record:

Col. 1	H
Col. 2–7	Program name
Col. 8–13	Starting address of object program (hexadecimal)
Col. 14–19	Length of object program in bytes (hexadecimal)

1	2	3	4	5	6	7	8	9	10
H	C	O	P	Y			0	0	1

11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	2	4

Object Program

Text record:

Col. 1	T
Col. 2–7	Starting address for object code in this record(hexadecimal)
Col. 8–9	Length of object code in this record in bytes (hexadecimal)
Col. 10–69	Object code, represented in hexadecimal (2 columns per byte of object code)

1	2	3	4	5	6	7	8	9	10
T	0	0	1	0	0	0	1	E	1

11	12	13	14	15	16	17	18	19	20
4	1	0	2	1	4	8	2	0	3

21	22	23	24	25	26	27	28	29	30
9	0	0	1	0	1	E	2	8	1

Text Record

31	32	33	34	35	36	37	38	39	40
0	2	1	3	0	1	0	1	5	4

41	42	43	44	45	46	47	48	49	50
8	2	0	6	1	3	C	1	0	0

51	52	53	54	55	56	57	58	59	60
3	0	0	1	0	1	B	0	C	1

61	62	63	64	65	66	67	68	69
0	1	E	4	5	4	F	4	6

End Record

End record:

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

1	2	3	4	5	6	7
E	0	0	1	0	0	0

Program Blocks and Control Sections

- Although the source program logically contains subroutines, data area, etc, they were assembled into a single block of object code in which the machine instructions and data appeared in the same order as they were in the source program.
- To provide flexibility:
  - Program blocks
    - Segments of code that are rearranged within a single object program unit
  - Control sections
    - Segments of code that are translated into independent object program units

Program Blocks

- As an *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
- The assembler directive USE indicates which portions of the source program belong to the various blocks.

Program with Multiple Program Blocks

At the beginning, the default block is assumed.

```
  5   COPY      START   0              COPY FILE FROM INPUT TO OUTPUT
 10   FIRST     STL     RETADR         SAVE RETURN ADDRESS
 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     =C'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
 92             USE     CDATA
 95   RETADR    RESW    1
100   LENGTH    RESW    1              LENGTH OF RECORD
103             USE     CBLKS
105   BUFFER    RESB    4096           4096-BYTE BUFFER AREA
106   BUFEND    EQU     *              FIRST LOCATION AFTER BUFFER
107   MAXLEN    EQU     BUFEND-BUFFER  MAXIMUM RECORD LENGTH
110
```

Program with Multiple Program Blocks

Resume the default block

```
110   .
115   .         SUBROUTINE TO READ RECORD INTO BUFFER
120   .
123             USE
125   RDREC     CLEAR     X                CLEAR LOOP COUNTER
130             CLEAR     A                CLEAR A TO ZERO
132             CLEAR     S                CLEAR S TO ZERO
133            +LDT      #MAXLEN
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
183             USE       CDATA
185   INPUT     BYTE      X'F1'            CODE FOR INPUT DEVICE
195
```

Resume the CDATA block

Program with Multiple Program Blocks

Resume the default block

```
.
.        SUBROUTINE TO WRITE RECORD FROM BUFFER
.
         USE
WRREC    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
         USE      CDATA
         LTORG
         END      FIRST
```

Resume the CDATA block

Program Blocks

- Each program block may actually contain several separate segments of the source program.
- The assembler will logically rearrange these segments to gather together the pieces of each block.
- The result is the same as if the programmer had physically rearranged the source statements to group together all the source lines belonging to each block.

Why Program Blocks

- To satisfy the contradictive goals:
  - Separate the program into blocks in a particular order
    - Large buffer area is moved to the end of the object program
    - Using the extended format instructions or base relative mode may be reduced. (lines 15, 35, and 65)
    - Placement of literal pool is easier: simply put them before the large data area, CDATA block. (line 253)
  - Data areas are scattered
    - Program readability is better if data areas are placed in the source program close to the statements that reference them.

How to Rearrange Codes into Program Blocks

- Pass 1
  - Maintain a separate LOCCTR for each program block
    - initialized to 0 when the block is first begun
    - saved when switching to another block
    - restored when resuming a previous block
  - Assign to each label an address relative to the start of the block that contains it
  - Store the block name or number in the SYMTAB along with the assigned relative address of the label
  - Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
  - Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

How to Rearrange Codes into Program Blocks

- Pass 2
  - Calculate the address for each symbol <u>relative to the start of the object program</u> by adding
    - the location of the symbol relative to the start of its block
    - the assigned starting address of this block

Object Program with Multiple Program Blocks

	Loc/Block					
5	0000	0	COPY	START	0	
10	0000	0	FIRST	STL	RETADR	172063
15	0003	0	CLOOP	JSUB	RDREC	4B2021
20	0006	0		LDA	LENGTH	032060
25	0009	0		COMP	#0	290000
30	000C	0		JEQ	ENDFIL	332006
35	000F	0		JSUB	WRREC	4B203B
40	0012	0		J	CLOOP	3F2FEE
45	0015	0	ENDFIL	LDA	=C'EOF'	032055
50	0018	0		STA	BUFFER	0F2056
55	001B	0		LDA	#3	010003
60	001E	0		STA	LENGTH	0F2048
65	0021	0		JSUB	WRREC	4B2029
70	0024	0		J	@RETADR	3E203F
92	0000	1		USE	CDATA	
95	0000	1	RETADR	RESW	1	
100	0003	1	LENGTH	RESW	1	
103	0000	2		USE	CBLKS	
105	0000	2	BUFFER	RESB	4096	
106	1000	2	BUFEND	EQU	*	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	
110						

0: default
1: CDATA
2: CBLKS

No block number because MAXLEN is an absolute symbol

Object Program with Multiple Program Blocks

```
115                         .         SUBROUTINE TO READ RECORD INTO BUFFER
120                         .
123   0027  0                         USE
125   0027  0   RDREC       CLEAR     X                 B410
130   0029  0               CLEAR     A                 B400
132   002B  0               CLEAR     S                 B440
133   002D  0               +LDT      #MAXLEN           75101000
135   0031  0   RLOOP       TD        INPUT             E32038
140   0034  0               JEQ       RLOOP             332FFA
145   0037  0               RD        INPUT             DB2032
150   003A  0               COMPR     A,S               A004
155   003C  0               JEQ       EXIT              332008
160   003F  0               STCH      BUFFER,X          57A02F
165   0042  0               TIXR      T                 B850
170   0044  0               JLT       RLOOP             3B2FEA
175   0047  0   EXIT        STX       LENGTH            13201F
180   004A  0               RSUB                        4F0000
183   0006  1               USE       CDATA
185   0006  1   INPUT       BYTE      X'F1'             F1
```

Object Program with Multiple Program Blocks

```
195                    .
200                    .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205                    .
208     004D  0                USE
210     004D  0    WRREC       CLEAR     X             B410
212     004F  0                LDT       LENGTH        772017
215     0052  0    WLOOP       TD        =X'05'        E3201B
220     0055  0                JEQ       WLOOP         332FFA
225     0058  0                LDCH      BUFFER,X      53A016
230     005B  0                WD        =X'05'        DF2012
235     005E  0                TIXR      T             B850
240     0060  0                JLT       WLOOP         3B2FEF
245     0063  0                RSUB                    4F0000
252     0007  1                USE       CDATA
253                            LTORG
        0007  1    *          =C'EOF'                  454F46
        000A  1    *          =X'05'                   05
255                            END       FIRST
```

Table for Program Blocks

- At the end of Pass 1:

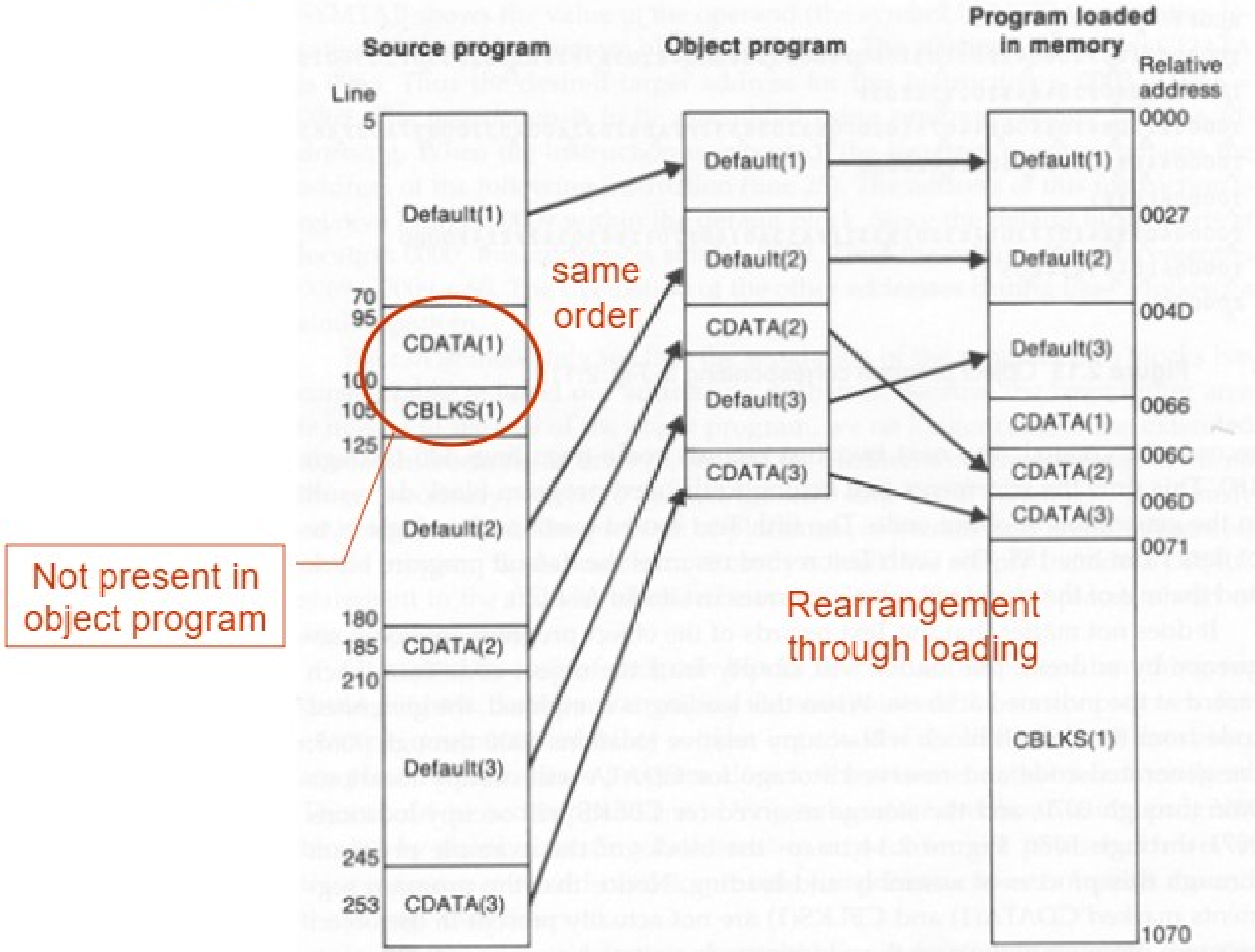
Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

Object Program

- It is not necessary to physically rearrange the generated code in the object program to place the pieces of each program block together.
- The assembler just simply insert the proper load address in each Text record.

```
H^COPY  ^000000^001071
T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEE^032055^0F2056^010003
T^00001E^09^0F2048^4B2029^3E203F
T^000027^1D^B410^B400^B440^75101000^E32038^332FFA^DB2032^A004^332008^57A02F^B850
T^000044^09^3B2FEA^13201F^4F0000
T^00006C^01^F1
T^00004D^19^B410^772017^E3201B^332FFA^53A016^DF2012^B850^3B2FEF^4F0000
T^00006D^04^454F46^05
E^000000
```

Program Blocks Loaded in Memory



Control Sections

- A control section
  - is a part of the program that maintains its identity after assembly
  - is often used for subroutine or other logical subdivision of a program
  - can be assembled, loaded, and relocated independently
  - is more flexible

Program Linking

- Program linking is used to link together logically related control sections
- Problem:
  - The assembler does not know where any other control section will be located at execution time.
  - When an instruction needs to refer to instructions or data located in another control section, the assembler is unable to process this reference.
  - The assembler has to generate information for such kind of references, called external references, that will allow the loader to perform the required linking.

Program with Multiple Control Sections

Implicitly defined as an external symbol

First control section: COPY

Define external symbols

External reference

```
COPY     START    0                COPY FILE FROM INPUT TO OUTPUT
         EXTDEF   BUFFER,BUFEND,LENGTH
         EXTREF   RDREC,WRREC
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
RETADR   RESW     1
LENGTH   RESW     1                LENGTH OF RECORD
         LTORG
BUFFER   RESB     4096             4096-BYTE BUFFER AREA
BUFEND   EQU      *
MAXLEN   EQU      BUFEND-BUFFER
```

Program with Multiple Control Sections

Implicitly defined as an external symbol

Second control section: RDREC

External reference

```
109   RDREC    CSECT
110   .
115   .        SUBROUTINE TO READ RECORD INTO BUFFER
120   .
122            EXTREF   BUFFER,LENGTH,BUFEND
125            CLEAR    X                  CLEAR LOOP COUNTER
130            CLEAR    A                  CLEAR A TO ZERO
132            CLEAR    S                  CLEAR S TO ZERO
133            LDT      MAXLEN
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
190   MAXLEN   WORD     BUFEND-BUFFER
```

Program with Multiple Control Sections

Implicitly defined as an external symbol

Third control section: WRREC

External reference

```
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 Directives for Control Section

- START:
  - start the first control section
  - set program name as the control section name
  - define the control section name as an external symbol
- CSECT:
  - start a new control section
  - specify the control section name
  - define the control section name as an external symbol
- EXTDEF:
  - define external symbols
- EXTREF:
  - name symbols defined in other control sections

How to Handle External References

15 0003 CLOOP +JSUB RDREC 4B100000

- The operand RDREC is an external reference.
- The assembler
  - has no idea where RDREC is
  - inserts an address of zero
  - can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)
  - passes information to the loader

How to Handle External References

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

- There are two external references in the expression, BUFEND and BUFFER.
- The assembler
  - inserts a value of zero
  - passes information to the loader
    - Add to this data area the address of BUFEND
    - Subtract from this data area the address of BUFFER
- On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

107 1000 MAXLEN EQU BUFEND-BUFFER

Object Code with Multiple Control Sections

5	0000	COPY	START	0	
6			EXTDEF	BUFFER,BUFEND,LENGTH	
7			EXTREF	RDREC,WRREC	
10	0000	FIRST	STL	RETADR	172027
15	0003	CLOOP	+JSUB	RDREC	4B100000
20	0007		LDA	LENGTH	032023
25	000A		COMP	#0	290000
30	000D		JEQ	ENDFIL	332007
35	0010		+JSUB	WRREC	4B100000
40	0014		J	CLOOP	3F2FEC
45	0017	ENDFIL	LDA	=C'EOF'	032016
50	001A		STA	BUFFER	0F2016
55	001D		LDA	#3	010003
60	0020		STA	LENGTH	0F200A
65	0023		+JSUB	WRREC	4B100000
70	0027		J	@RETADR	3E2000
95	002A	RETADR	RESW	1	
100	002D	LENGTH	RESW	1	
103			LTORG		
	0030	*	=C'EOF'		454F46
105	0033	BUFFER	RESB	4096	
106	1033	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	

Object Code with Multiple Control Sections

```
109   0000   RDREC    CSECT
110                   .
115                   .        SUBROUTINE TO READ RECORD INTO BUFFER
120                   .
122                   EXTREF   BUFFER,LENGTH,BUFEND
125   0000            CLEAR    X                    B410
130   0002            CLEAR    A                    B400
132   0004            CLEAR    S                    B440
133   0006            LDT      MAXLEN               77201F
135   0009   RLOOP    TD       INPUT                E3201B
140   000C            JEQ      RLOOP                332FFA
145   000F            RD       INPUT                DB2015
150   0012            COMPR    A,S                  A004
155   0014            JEQ      EXIT                 332009
160   0017           +STCH     BUFFER,X             57900000
165   001B            TIXR     T                    B850
170   001D            JLT      RLOOP                3B2FE9
175   0020   EXIT    +STX      LENGTH               13100000
180   0024            RSUB                          4F0000
185   0027   INPUT    BYTE     X'F1'                F1
190   0028   MAXLEN   WORD     BUFEND-BUFFER        000000
```

Object Code with Multiple Control Sections

```
193   0000   WRREC   CSECT
195          .
200          .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205          .
207                  EXTREF  LENGTH,BUFFER
210   0000           CLEAR   X                 B410
212   0002          +LDT     LENGTH            77100000
215   0006   WLOOP   TD     =X'05'             E32012
220   0009           JEQ     WLOOP             332FFA
225   000C          +LDCH    BUFFER,X          53900000
230   0010           WD     =X'05'             DF2008
235   0013           TIXR    T                 B850
240   0015           JLT     WLOOP             3B2FEE
245   0018           RSUB                      4F0000
255                  END     FIRST
      001B   *      =X'05'                     05
```

How to Handle Control Sections

- The assembler
  - processes each control section independently
  - establishes a separate LOCCTR (initialized to 0) for each control section
  - stores in SYMTAB the control section in which a symbol is defined
  - allow the same symbol to be used in different control sections
  - reports an error when attempting to refer to a symbol in another control section, unless the symbol is defined as an external reference
  - generates information in the object program for external references

New Records for External References

Define record: gives information about external symbols named by EXTDEF

Col. 1	D
Col. 2–7	Name of external symbol defined in this control section
Col. 8–13	Relative address of symbol within this control section (hexadecimal)
Col. 14–73	Repeat information in Col. 2–13 for other external symbols

Refer record: lists symbols used as external references, i.e., symbols named by EXTREF

Col. 1	R
Col. 2–7	Name of external symbol referred to in this control section
Col. 8–73	Names of other external reference symbols

Revised Modification Record

Modification record (revised):

Col. 1	M
Col. 2–7	Starting address of the field to be modified, relative to the beginning of the control section (hexadecimal)
Col. 8–9	Length of the field to be modified, in half-bytes (hexadecimal)
Col. 10	Modification flag (+ or –)
Col. 11–16	External symbol whose value is to be added to or subtracted from the indicated field

Object Program

COPY

```
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B1000000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000
```

RDREC

```
HRDREC 000000 00002B
RBUFFERLENGTHBUFEND
T000000 1D B410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D 0E 3B2FE9131000004F0000F1000000
M000018 05+BUFFER
M000021 05+LENGTH
M000028 06+BUFEND
M000028 06-BUFFER
E
```

WRREC

```
HWRREC 000000 00001C
RLENGTHBUFFER
T000000 1C B410 77100000 E32012 332FFA 53900000 DF2008 B850 3B2FEE 4F000005
M000003 05+LENGTH
M00000D 05+BUFFER
E
```