System Programming 2023 Spring Solution

1.a. Explain the architecture of the SIC and SIC/XE machines with important features.

Simplified Instructional Computer (SIC) is a hypothetical computer that has hardware features that are often found in real machines. There are two versions of this machine:

- SIC standard Model
- SIC/XE(extra equipment or expensive)

SIC Machine Architecture

i. Memory

- Memory is byte-addressable i.e., words are addressed by the location of their lowest-numbered byte.
- There are 2^15 bytes in computer memory (1 byte = 8 bits)
- 3 consecutive byte = 1 word (24 bits = 1 word)
- There are a total of 32,768 (2^15) bytes

ii. Registers

There are 5 registers in SIC. Every register has an address associated with it known as a registration number. The size of each register is 3 bytes. Based on register size, integer size is dependent.

Mnemonic	**Number**	**Name**	**Purpose**
A	0	Accumulator	It is used for mathematical operations.
X	1	Index register	It is used for addressing
L	2	Linkage register	It stores the return address of the instruction in case of subroutines.
PC	8	Program Counter	It contains the address of the next instruction to be fetched for execution
SW	9	Status Word	Contains a variety of information including

			Condition Code(CC)

iii. Data Formats

- Integers are stored as 24-bit binary numbers
- 2's compliment representation is used for negative values
- Characters are stored using 8-bit ASCII codes
- Floating point is not supported

iv. Instruction Formats

All machine instructions on the standard SIC have 24-bit instruction format

Opcode	x	Address

Opcode = 8 bits Address = 15 bits
x(Flag bit) = 1 bit

- If x=0 it means direct addressing mode.
- If x=1 it means indexed addressing mode.

v. Addressing Mode

There are two addressing modes available, indicated by the setting of the x bit in the instruction

Mode	Indication	TA Calculation
Direct	x = 0	TA = Address
Indexed	x = 1	TA = Address + (X)

(X) represents the contents of register X

vi. Instruction Set

- Load And Store Instructions: To move or store data from accumulator to memory or vice-versa. For example LDA, STA, LDX, STX, etc.
- Comparison Instructions: Used to compare data in memory by contents in accumulator. For example COMP data.
- Arithmetic Instructions: Used to perform operations on accumulator and memory and store results in the accumulator. For example ADD, SUB, MUL, DIV, etc.
- Conditional Jump: compare the contents of accumulator and memory and performs task based on conditions. For example JLT, JEQ, JGT
- Subroutine Linkage: Instructions related to subroutines. For example JSUB, RSUB

vii. Input and Output

On the standard version of SIC, input, and output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A. Each device is assigned a unique 8-bit code. There are three I/O instructions, each of which specifies the device code as an operand.

SIC/XE Machine Architecture

i. Memory

- Maximum memory available is 1 megabyte (2^20 byte)
- 3 consecutive byte = 1 word (24 bits = 1 word)

ii. Registers

It contain 9 registers (5 SIC registers + 4 additional registers).

Mnemonic	**Number**	**Name**	**Purpose**
A	0	Accumulator	It is used for mathematical operations.
X	1	Index register	It is used for addressing
L	2	Linkage register	It stores the return address of the instruction in case of subroutines.
B	3	Base register	It is also used for addressing.
S	4		General working register; no special use
T	5		General working register; no special use
F	6	Floating point	It is used to store

		accumulator	floating point numbers.
PC	8	Program Counter	It contains the address of the next instruction to be fetched for execution
SW	9	Status Word	Contains a variety of information including Condition Code(CC)

*** Floating point accumulator = 48 bits*

iii. Data Formats

- SIC/XE provides the same data formats as the standard version.
- It has extra 48-bit floating point data type

1	11	36
s	Exponent (e)	Fraction (f)

s= 0 for positive number
s=1 for negative number

The absolute value of the number is represented as

$$f * 2^{(e-1024)}$$

iv. Instruction Formats

In SIC/XE architecture there are 4 types of formats available

- Format 1 (1 byte)

opcode (8 bits)

- Format 2 (2 bytes)

Opcode (8 bits)	r1 (4 bits)	r2 (4 bits)

Example: COMPR A,S

- Format 3 (3 bytes)

Opcode (6 bits)	n	i	x	b	p	e	Displacement (12 bits)

Example: LDA #3

n=*Indirect mode*, i=*Immediate addressing,* x=*Index addressing*, b=*Base addressing*, p=*Program counter*, e=*extended addressing*

- Format 4 (4 bytes)

Opcode (6 bits)	n	i	x	b	p	e	Address (20 bits)

Example: +JSUB ALPHA

v. Addressing Modes

Two new relative addressing modes are available for the use with instructions assembled using Format 3

Mode	Indication	Target Address Calculation
Base Relative	b=1,p=0	TA = (B) + disp ($0 \leq disp \leq 4095$)
Program counter relative	b=0, p=1	TA = (PC) + disp ($-2048 \leq disp \leq 2047$)

vi. Instruction Set

In SIC/XE all the instructions are the same as that of SIC architecture but because of the Floating point data format, it provides Floating point Arithmetic functions too.

To perform floating-point arithmetic operations,

ADDF	Add floating point numbers
SUBF	Subtract floating point numbers
MULF	Multiply floating point numbers
DIVF	Divide floating point numbers

vii. Input and Output
SIC/XE architecture includes I/O channels that allow to perform I/O operations while CPU is executing other tasks. It will enable overlapping of computing and I/O, which make this architecture more efficient.

1.b. Write a SIC program for arithmetic operations
Lets perform the following operations:
BETA = ALPHA + INCR - ONE
Where, ALPHA, BETA and INCR are integer variables and ONE = 1
*** you can choose any other expression ***

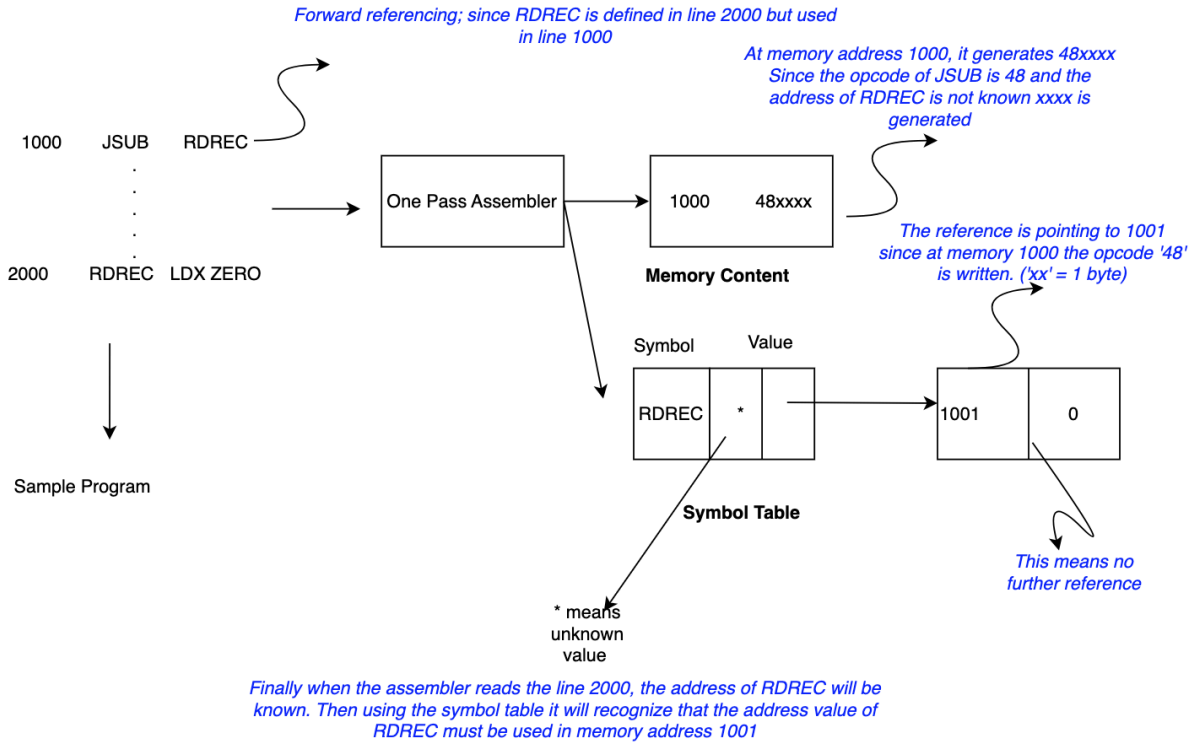
	LDA	ALPHA	Load ALPHA into register A
	ADD	INCR	Add the value of INCR and store the result in register A
	SUB	ONE	Subtract the value of ONE from the value already stored in register A; also saves the resulting value in A
	STA	BETA	Store the value of register A in BETA
	. . .		
ONE	WORD	1	one word constant i.e; ONE =1
ALPHA	RESW	1	One-word variable
BETA	RESW	1	One-word variable
INCR	RESW	1	One-word variable

2.a. How is the forward reference handled in the one pass assembler?
A single pass assembler is a type of assembler that can generate machine code in a single pass over the source code. This means that it processes the source code only once, without the need for multiple passes.
An assembler can generate object code only when the address of the variable used in the program is known. One of the challenges with a single-pass assembler is handling forward references, which occur when a symbol is used before it is defined in the source code.

If forward referencing is used in a souce program the assembler will do the following:

1. The assembler generates object code instructions as it scans the source program.
2. If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled.
3. The symbol used as an operand is entered into the symbol table (unless such an entry is already present)
4. The entry is flagged to indicate that the symbol is undefined.
5. The address of the operand field of the instruction that refers to the undefined symbol is added to the list of forward references associated with the symbol table entry.
6. When the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if one exists)
7. Finally, the proper address is inserted into any instructions previously generated.

Example



[For more details, refer to page number 98 and 99 of the text book]

2.b. Explain machine-dependent assembler features.

Machine-dependent assembler features are

- Instruction formats and Addressing modes
- Program relocation

Instruction Formats and Addressing Modes

Addressing Mode	Format
Immediate Addressing Mode	opcode #c
Indirect Addressing Mode	opcode @m
PC relative or Base relative ● The assembler directive BASE is used with base-relative addressing ● If displacements are too large to fit into a 3-byte instruction, then 4-byte extended format is used	opcode m
Extended Format	+opcode m
Indexed Addressing	opcode m,X
Register to Register instructions	

Advantages:
- Support multiprogramming and need program reallocation capability
- Improve the execution speed

Program Relocation
The larger main memory of SIC/XE means that several programs can be loaded and run at the same time. This kind of sharing of the machine between programs is
called multiprogramming. To take full advantage of the assembler's capabilities, the following things should be done:
- Load programs into memory wherever there is room
- Not specifying a fixed address at assembly time

This can be achieved by using the program relocation feature.
Program Relocation Program relocation refers to the process of adjusting the memory addresses of a program during the loading or execution phase.

For example:

Absolute program (or *absolute assembly*)

- Program must be loaded at the address specified *at assembly time.*
- E.g. Fig. 2.1

```
COPY   START   1000
FIRST  STL     RETADR
                :
                :
```

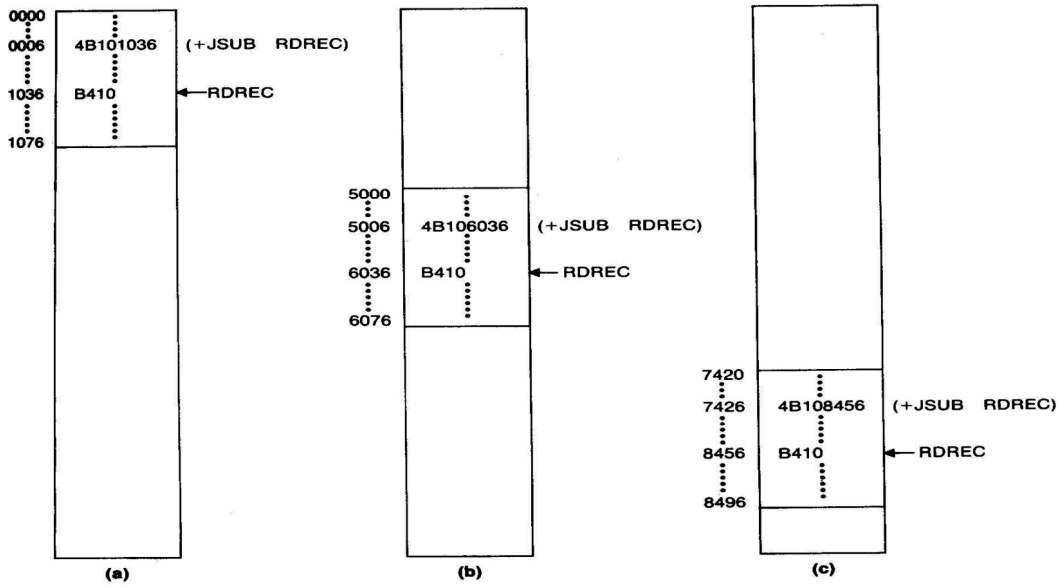
program loading starting address 1000

  - e.g. 55 101B LDA THREE 00102D

- What if the program is loaded to 2000

  e.g. 55 101B LDA THREE 00202D

  - Each absolute address should be modified



Example of Program Relocation

Relocatable program

```
COPY   START   0
FIRST  STL     RETADR
         :
         :
```

program loading starting address is determined *at load time*

- An object program that contains the information necessary to perform *address modification* for relocation
- The **assembler** must identify for the **loader** those parts of object program that need modification.
- No instruction modification is needed for
  - Immediate addressing (not a memory address)
  - PC-relative, Base-relative addressing
- The only parts of the program that require modification at load time are those that specify *direct addresses*
  - In SIC/XE, only found in extended format instructions

56

Modification record

- Col 1 M
- Col 2-7 Starting location of the address field to be modified, relative to the beginning of the program (hex)
- Col 8-9 length of the address field to be modified, in half-bytes
- E.g M^000007^05

3.a. Consider the following assembly program.

- Fill column for location counter
- Create object code column with object codes
- Show all data structures
- Create Object code file

Line	LOCCTR	Symbol	Opcode	Exp	Object Code
10	0000	Test	START	0	
20			EXTDEF	Odev	
30			EXTREF	Ch,Phash	
40	0000	Begin	LDA	=C'A'	032008

50	0003		+STA	Ch	0F100000
60	0007		+JSUB	Phash	4B100000
70			LTORG		
	000B	*	=C'A'		41
80	000C	Odev	BYTE	X'06'	06
90	0000	Phash	CSECT		
100			EXTDEF	Ch	
110			EXTREF	Odev	
120	0000	Loop	+TD	Odev	E0100000
130	0004		JEQ	Loop	332FF9
140	0007		LDCH	Ch	532007
150	000A		+WD	Odev	DF100000
160	000E		RSUB		4C0000
170	0011	Ch	RESB	10	
180			END	Begin	

*Explainations (** **Not Required in Exam** **)*

- *This program is an example of Control Section and Program Linking.*
- *EXTDEF means External Definition, which means the symbol is defined in the section (used in line number 20 and 100)*
- *EXTREF means External Reference, which means the symbol is used in the section but defined elsewhere (used in line 30 and 110). For example, in the first control section (i.e from line 10 to 80) Ch(line 50) and Phash(line 60) is used but those symbols are defined in different control section (line 90 to 180).*
- *EXTREF and EXTDEF are directives and no line counter should be specified to them.*
- *CSECT (used in line 90) is also an directive which means the start of a new section.*
- *There are two sections in this program, the first section is from line 10 to line 80 and another section(named Phash) from line 90.*
- *CSECT doesn't require Location Counter.*

- *When a new section starts, the location counter will start back from 0000 (line number 90)*
- *In line 40, Literal is used (defined by =C'A'). This has to be defined in a literal pool.*
- *A literal pool can be defined either after the END of the program or after using LTORG directive.*
- *In above example, LTORG is defined in line 70, hence we have to define the literal pool just below that line(highlighted in green)*
- *Do not confuse the declaration of X'06' in line 80 as a literal. It is NOT A LITERAL.*
- *A literal must have '=' symbol.*

Object Code Generation Explanation [You have to show these calculations in exam]

- Line number 10,20, and 30 are just directives; hence no object code will be generated.
- Line number 40

  LDA =C'A

  Since the expression=C'A' is a literal, we have defined it in a literal pool below LTORG after line 70. The address of the literal is 000B (highlighted in GREEN)

  Given; LDA : 00

						n	i	x	b	p	e	
0	0	0	0	0	0	1	1	0	0	1	0	Displacement

  Displacement = 000B - 0003 = 0008
  n & i = 1; since it is neither indirect nor immediate
  x = 0; since it is not indexed addressing
  b = 0 ; since it is not Base relative
  p =1; since it is PC relative because the displacement is in the range -2048 to 2047
  e = 0; since it is not format 4 instruction

So finally, the object code is 032008

- Line 50

  +STA Ch

  It is a format 4 instruction
  Since Ch is an external reference, the assembler wont have the address of Ch, so we have to write 00000 in the place of Ch's address

  STA : 0C

						n	i	x	b	p	e	
0	0	0	0	1	1	1	1	0	0	0	1	Address = 00000

Here,
n & i = 1; since it is neither indirect nor immediate
x = 0; since it is not indexed addressing
b = 0 ; since it is not Base relative
p =0; since it is not PC relative either (Format 4 instruction uses absolute addressing)
e = 1; since it is a format 4 instruction
Address = 00000; since we Ch is defined in another CSECT.

Hence, the object code is 0F100000

- Line 70 is same as Line 60
- Line 80 is a single byte declaration of value 06 hence the object code is 06
- After Line 70 and before Line 80, we have declared a literal pool. The object code is 41 because the HexCode of A is 41
- Line 130 is same as Line 70 and 60
- Line 140
  JEQ LOOP
  Here the label LOOP is declared before this line. In such cases, if you try to find the displacement the result will be a negative number.

  JEQ= 30

						n	i	x	b	p	e	
0	0	1	1	0	0	1	1	0	0	1	0	Displacement

  Displacement = Address of LOOP - 0007 = 0000 - 0007 = -7
  So the 2's complement of -7 is FF9
  So the object code is 332FF9
- The remaining lines are all the same.

The data structure of the above program contains the following:

- LOCCTR (location counter; already filled up above)
- OPTAB (Always given in question)
- SYMTAB (list of symbols used in the program)
- LITTAB (table of literals used in the program)

Symbol Table(SYMTAB) [** *You can also divide this table into 2 separate tables, one for each section*]

Symbol	Address/Value
Test	0000
Begin	0000
Odev	000C
Phas	0000
Loop	0000
Ch	0011

Literal Table(LITTAB)

Literal	Hex Value	Length	Address
C'A'	41	1	00B

Object Code [*There are two sections so there will be two object files*]

H□Test □000000□00000D
D□Odev □00000C
R□Ch □Phash
T□000000□0D□032008□0F100000□4B100000□41□06
M□000004□05□+Ch
M□000008□05□+Phash
E□0000000

H□Phash □000000□000022
D□Ch □000011
R□Odev
T□000000□11□E0100000□332FF9□532007□DF100000□4C0000
M□000000□05□+Odev
M□00000A□05□+Odev
E□0000000

3.b. What is absolute loader? Explain with proper algorithm.

The absolute loader transfers the text of the program into memory at the address provided by the assembler after reading the object program line by line. There are two types of information that the object program must communicate from the assembler to the loader.

- It must convey the machine instructions that the assembler has created along with the memory address.
- It must convey the start of the execution. At this point, the software will begin to run after it has loaded.

The object program is the sequence of the object records. Each object record specifies some specific aspect of the program in the object module. There are two types of records:

- Text record containing a binary image of the assembly program.
- Transfer the record that contains the execution's starting or entry point.

Advantages of Absolute Loader

- Simple and efficient

Disadvantages

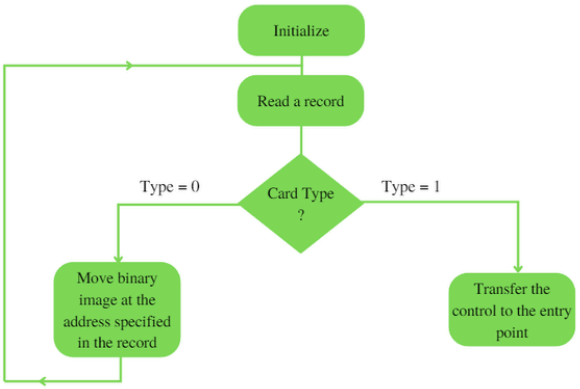
- The need for the programmer to specify the actual address at which it will be loaded into memory
- Difficult to use subroutine libraries efficiently

Absolute loader only performs loading function. It does not need to perform linking and program relocation. All functions are accomplished in a single pass.

Design of Absolute Loader (Only has Single Pass)

1. Check the Header record for program name, starting address, and length
2. Bring the object program contained in theText record to the indicated address
3. No need to perform program linking and relocation
4. Start the execution by jumping to the address specified in the End record

Flowchart of Absolute Loader (Not necessary)



Algorithm *[** This is essential]*

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

E.g., convert the pair of characters "14" (two bytes) in the object program to a single byte with hexadecimal value 14

4.

4. Consider the following assembly language program.

Line	Symbol	Opcode	Exp
10	TEST	START	2050
20	FIRST	LDA	P
30		MUL	RATE
40		MUL	TIME
50		DIV	HUNDRED
60		STA	SI
70	P	RESW	1
80	RATE	RESW	1
90	TIME	RESW	1
100	HUNDRED	WORD	100
110	SI	RESW	1
120		END	FIRST

Mnemonic	Opcode
LDA	00
MUL	20
DIV	24
STA	0C

i. Fill column for location counter
ii. Create object code column with object codes
iii. Create Object code file.
iv. Load the program in memory

The above program is an SIC program since there are no '+' symbols, no literals or '@' symbols.

Line	**LOCTR**	**Symbol**	**Opcode**	**Exp**	**Object Code**
10	2050	Test	Start	2050	
20	2050	First	LDA	P	00205F
30	2053		MUL	RATE	202062
40	2056		MUL	TIME	202065
50	2059		DIV	HUNDRED	242068
60	205C		STA	SI	0C206B
70	205F	P	RESW	1	
80	2062	RATE	RESW	1	

90	2065	TIME	RESW	1	
100	2068	HUNDRED	WORD	100	000064
110	206B	SI	RESW	1	
120			END	FIRST	

In the above table (line 100), 100 in decimal is 64 in Hex, hence the object code ie 000064.
The generation of object code of this program is very straightforward, so I have omitted the calculation part. But you need to do it in the exam

Object Code File

H□Test □002050□00001E
T□12□00205F□202062□202065□242068□0C206B□000064
E□002050

Program Loaded into Memory

Memory Address	**Contents**			
0000	XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX			
0010	XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX			
.				
.				
.				
2050	00205F20	20622020	65242068	0C206BXX
2060	XXXXXXXX XXXXXXXX		000064XX	XXXXXXX

** XX = 1 BYTE i.e X = half byte
Since there are no object code after 205C up until 2068, we have to fill in XX…XX

5.a. What is macro time variable? How macro processor manages value of macro time variable.

Macro time variable refers to a variable that is evaluated and used during the macro expansion process, which occurs before the actual program execution.Macro time variables are integral to the operation of macro processors, helping to tailor the output generated by macros based on conditions or calculations performed during the macro expansion phase.

- Macro time variables are processed when the macro is being expanded, not during the execution of the generated code. This means they are only relevant during the compilation or assembly phase.
- These variables are often used to control the flow of macro expansion, generate code conditionally, or perform calculations that influence how the macro generates code.
- They are mostly used for conditional compilations, counters and symbol replacement.

Macro processors manage the value of macro time variables by handling them directly during the macro expansion process. Here's how the macro processor manages these variables:

1. Definition and Initialization:
- **Definition**: Macro time variables are typically defined within the macro definition itself. They can be initialized with a specific value or set dynamically during macro expansion.
- **Initialization:** These variables are initialized either at the start of the macro expansion or whenever a specific macro statement assigns a value to them.

2. Storage and Scope:
- **Storage:** The macro processor maintains a symbol table or an internal data structure (like a stack or list) to store macro time variables and their values.
- **Scope:** The scope of a macro time variable is usually limited to the macro in which it is defined. Once the macro expansion is complete, the variable is typically discarded unless it is explicitly passed or preserved for subsequent macro expansions.

3. Modification:
- **Assignment Statements:** Macro processors use assignment statements to change the value of macro time variables. For example, statements like SET, LET, or = are often used within macros to modify variable values.
- **Increment/Decrement:** Variables can be updated using arithmetic operations, such as incrementing or decrementing a counter.

4. Evaluation:
- **During Expansion:** The values of macro time variables are evaluated as the macro processor reads and expands the macro statements. If a macro references a variable, the processor fetches its current value from the symbol table and substitutes it into the expanded code.
- **Conditional Checks:** Values are often evaluated in conditional expressions (e.g., IF, WHILE) to control which parts of the macro are expanded.

5. Reset or Cleanup:
- **Resetting Values**: Macro time variables can be reset or reinitialized within the macro or explicitly cleared by the processor when the macro finishes expanding.

- **Cleanup:** After the macro expansion, the processor cleans up these variables, removing them from the symbol table or freeing any allocated memory, especially if they are not required for further expansions.

5.b Explain the concatenation of macro parameters.

Concatenation of macro parameters refers to the process of joining two or more macro parameters (or symbols) together to form a single continuous string or symbol during macro expansion. This feature is particularly useful in macro processors when you need to dynamically generate variable names, labels, or other symbols by combining fixed parts with varying parameters.

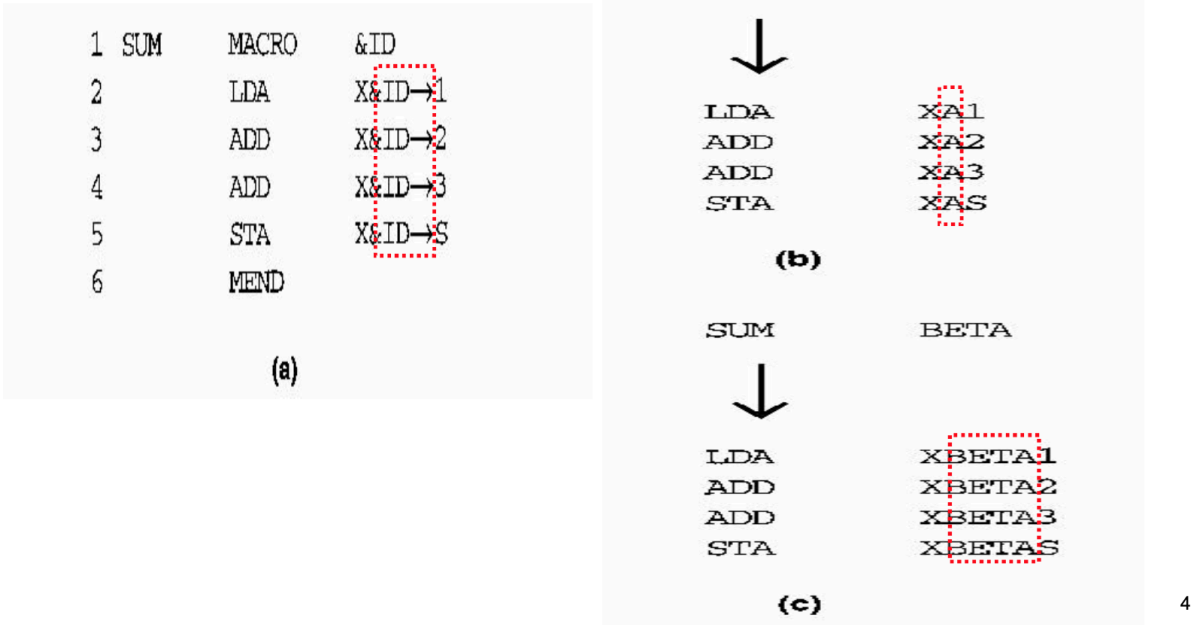
How Concatenation Works in Macro Processors

- Concatenation Operator: Most macro processors provide a specific operator or syntax for concatenation, such as the & symbol, which allows the programmer to combine macro parameters or literals.
- Combining Parameters: Concatenation allows you to create new symbols by merging parameter values with other text. For example, if a macro has parameters P1 and P2, concatenating them with & might result in a new symbol P1P2.
- Dynamic Symbol Creation: Concatenation is commonly used for creating dynamic variable names, labels, or other identifiers based on parameter values passed during macro invocation.

Usage of Concatenation

- Avoiding Name Conflicts: By creating unique symbols based on parameters, concatenation helps prevent naming conflicts in large assemblies or programs.
- Simplifying Code: It automates repetitive tasks, like generating a series of related variables or labels without manually typing each variation.

Example



4

5.c. Explain conditional macro expansion

Conditional macro expansion refers to the ability of macro processors to selectively expand or generate code based on certain conditions evaluated during the macro expansion phase. This feature allows macros to include or exclude parts of their code based on the values of parameters, macro-time variables, or specific conditions.

Conditional macro expansion typically uses specific directives or statements that evaluate conditions during the macro expansion phase. These conditions determine which parts of the macro are expanded and included in the final output.

IF, ELSE, ENDIF are common directives that allow the macro to conditionally include or exclude code based on a specified condition.

Example

```
MACRO CONDITIONAL_EXAMPLE PARAM
  IF PARAM EQ 1
    MOV AX, 1  // Code included if PARAM is 1
  ELSE
    MOV AX, 0  //Code included if PARAM is not 1
  ENDIF
MEND
```

Key Benefits of Conditional Expansion

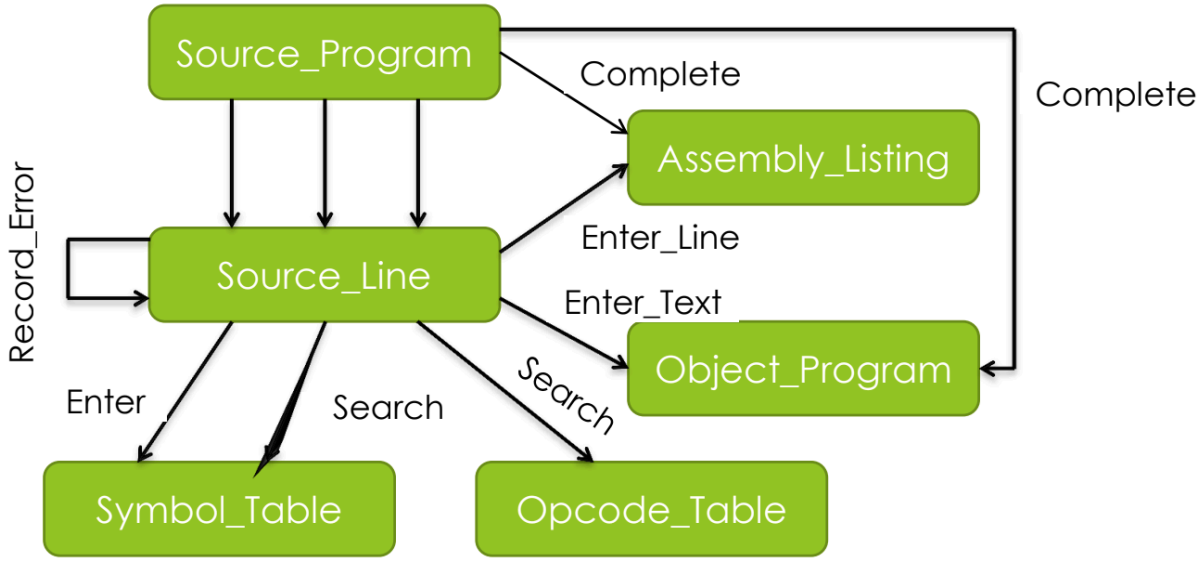
- Code Optimization: Conditional expansion enables the generation of optimized code paths depending on input values.
- Reduced Redundancy: It eliminates the need for multiple macros for slightly different tasks, as the same macro can adapt based on conditions.
- Dynamic Code Generation: Allows for dynamic decisions at the macro level, improving the flexibility of code.

6.a Explain the object diagram for the assembler with a diagram.

An object diagram for an assembler illustrates the structure and relationships between the key objects involved in the assembly process. Assemblers translate assembly language code into machine code, and the object diagram helps visualize how different components interact during this process.

Key Objects in an Assembler Object Diagram

- Source Program: This object represents the input assembly language program, containing instructions, labels, directives, and comments.
-
- Symbol Table: This object stores information about symbols (labels, variables) encountered during the assembly process. It maintains data like symbol names, addresses, and types, essential for resolving references during the second pass.
- Literal Table: This table keeps track of literals (constants) used in the source program. It stores the value, length, and assigned memory addresses.
- Opcode Table (OPTAB): This table contains the opcode mnemonics and their corresponding machine code equivalents. It helps the assembler translate mnemonic instructions into machine code.
- Intermediate File: This object holds the partially processed information generated during the first pass of a two-pass assembler, including the expanded instructions with addresses. It is used in the second pass to generate the final object code.
- Location Counter: This is a utility object used to keep track of memory addresses while assembling the program. It updates as the assembler processes each line of code, helping assign addresses to instructions and data.
- Assembler: The core object that coordinates the assembly process. It interacts with all other objects to read the source program, generate the symbol and literal tables, look up opcodes, and produce the object code.
- Object Code: This is the final output of the assembler, representing the machine code instructions that can be executed by the processor.
- Error Handler: This object identifies, manages, and reports errors encountered during the assembly process, such as undefined symbols, invalid opcodes, or syntax errors.



Explanation of Object diagram

- **Source Program → Assembler:** The assembler reads the source program, line by line.
- **Assembler ↔ Symbol Table**: As symbols (labels, variables) are encountered, the assembler updates the symbol table with their names, addresses, and attributes. The symbol table is referenced again in the second pass to resolve addresses.
- **Assembler ↔ Opcode Table (OPTAB):** The assembler uses the opcode table to translate mnemonic instructions into their binary equivalents.
- **Assembler ↔ Literal Table:** Literals found in the code are added to the literal table, which manages their placement in memory and assigns addresses.
- **Assembler → Intermediate File:** During the first pass, the assembler generates an intermediate file containing the expanded instructions, addresses, and placeholders for unresolved symbols.
- **Location Counter ↔ Assembler**: The location counter helps track the current memory address while processing instructions and data, ensuring correct address assignment.
- **Assembler → Object Code:** In the second pass, the assembler uses the information in the intermediate file, symbol table, and literal table to generate the final object code.
- **Assembler ↔ Error Handler:** The error handler is invoked whenever an error is detected, ensuring that errors are logged and reported to the user.