SYSTEM SOFTWARE

Module 1

Module-1 (Introduction) System Software vs Application Software, Different System Software– Assembler, Linker, Loader, Macro Processor, Text Editor, Debugger, Device Driver, Compiler, Interpreter, Operating System (Basic Concepts only). SIC & SIC/XE Architecture, Addressing modes, SIC & SIC/XE Instruction set , Assembler Directives.

Introduction

SOFTWARE

- Set of instructions given to the computer.
- We cannot touch and feel it.
- Developed by writing instructions in programming language.
- Operations of computer are controlled via this.
- If damaged or corrupted, back up copy can be installed again.
- Eg:- Antivirus, Microsoft Office Tools.

HARDWARE

- Physical parts of a computer.
- We can touch and feel it.
- Constructed using physical components.
- Operates under control of software.
- If damaged, can be replaced.
- Eg:- Keyboard, Monitor, Mouse

SOFTWARE vs HARDWARE

SOFTWARE	HARDWARE
1. Collection of instructions that tells computer what to do	1. Physical elements of computer
2. Divided in to a. System Software b. Application Software	2. Categories a. Input Devices. b. Output Devices

c. Utility Software	c. Storage Devices
3. Should be installed in to computer	3. Once software is loaded these can be used.
4. Prone to viruses	4. No virus attacks
5. If damaged/ corrupted reinstallation is possible	5. If damaged, can be replaced.
Eg:- Microsoft Office, Adobe	Eg:- Mouse, Monitor, Keyboard

TYPES OF SOFTWARE

1. System Software:

It is a collection of programs that bridges the gap between the level at which users wishes to interact with the computer and level at which computer is capable of operating. System software makes it possible for users to focus on applications with out needing to know the details of how the machine works internally.

- These are of different types

a) Operating System

b) Language Translators

   i. Compiler

   ii. Assembler

   iii. Interpreter

   iv. Macro Processor

c) Loader

d) Linker

e) Debugger

f) Text Editor

2.Application Software:

It refers to programs designed to perform specific tasks or functions for the user. Application software are typically built on top of an operating system, relying on the OS to provide necessary resources and services to run efficiently. Users interact with application software through graphical interfaces, command lines or any other methods depending on the software design.

- It allows end users to accomplish one or more specific tasks.
- Focus on application or problem to be solved.

Eg: Microsoft word,Adobe Photo shop,Google chrome,zoom

System Software vs Application software

System software	Application Software
Starts running when system is turned on	Runs when user requests to do so
No direct interaction with users	Interact with users directly
Written in low level languages and are complex programs	Written in high level languages and are easy programs compared to system software programs
A system cannot run or function any application without system software	It is user specific and not required to run the system
Usually comes pre installed with computer and is essential for booting up and functioning	Can be installed or uninstalled by the user based on their needs. Users can choose which applications to install and use.

Types of System Software

a) Operating System

An Operating system is the primary system software that manages all the hardware and all other software on the computer. It serves as an intermediatory between users and hardware ,ensuring all programs ca run properly by managing resources like the CPU,storage ,memory and input or output devices.

Eg: Windows,DOS,linux

Key functions include:

- Process Management
- Memory management
- File management
- Device management
- User interface
- Data management
- I/O operations
- Resource Management

b).Language Translators

a).Compiler

b).Assembler

c).Interpreter

d).Macro Processor

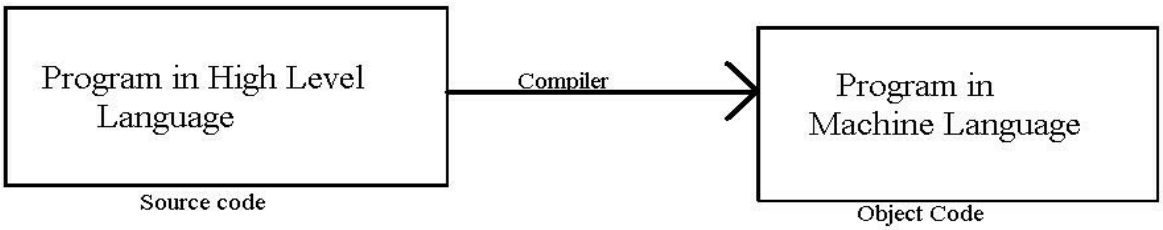
Language Translators

- Program that takes input program in one language and produces an output in another language.

I. Compilers

- Translates program written in high level language in to machine level language.
- Conversion or translation is taking place by taking program as whole.
- Bridges the semantic gap between language domain and execution domain.

  Eg : Programming Languages that use compilers are C,C++,JAVA,RUST,GO,SWIFT



The process of compiling a code involves several stages.

1.Lexical analysis

2.syntax analysis

3.Semantic analysis

4.Intermediate code generation

4.Code Optimization

5.Code Generation

Interpreters

- Translates statement of high level language in to machine level language by taking the **program line by line.**
- Interpretation cycle includes:

i) Fetch the statement.

ii) Analyze the statement and determine its meaning.

iii) Execute the meaning of statement.

Eg: The programming languages that use interpreters are JAVA Script,PYTHON,RUBY,PHP

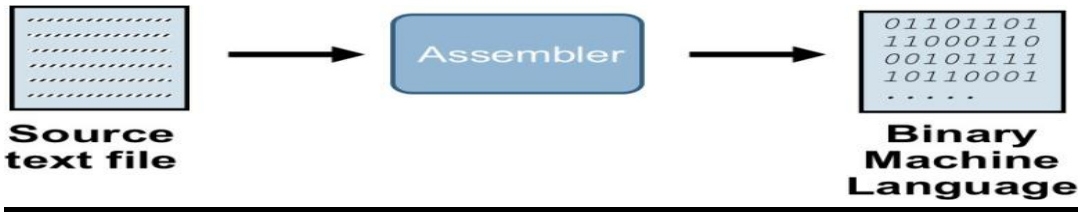
Compilers Vs Interpreters

compilers	**Interpreters**
Take the entire program for translation	Translates and executes the program line by line
Produce Intermediate code	No intermediate code is generated
Source code is translated into machine code before execution	
Generated Machine code which is specific to particular architecture.so it is less portable	Same source code can run on any platform that has an interpreter. More portable than compilers
Faster execution and errors are detected during compilation before running the program.	Slower execution compared to compilation and errors are detected at run time.

II. Assemblers

It is a tool that translates assembly languages which is a low level human readable (mnemonic codes) representation of machine code into actual machine code that a processor can execute. That means it involves direct mapping between symbolic instructions to machine code for execution.

- Programmers found it difficult to read or write programs in machine language, so for convenience they used mnemonic symbols for each instruction which is translated to machine language.
- Assemblers translate assembly language to machine language.
- Translate mnemonic code to machine language equivalents.
- Assign machine address to symbol table.



Compiler vs Assembler

Compilers	Assemblers
Translate high level languages into machine codes.	Translate assembly languages into machine codes
Complex programs as high level languages need several stages of translation.	Simpler process compared to compilation as assembly language translation is relatively straight forward.
Time consuming process compared to assemblers	Speed- usually fast because it involves direct translation
Eg : **GCC** (For C,C++,FORTRAN,GO) **Javac** for Java **Intel C++ compiler** (for C and C++)	Eg: MASM(Microsoft Macro assemblers) NASM(Netwide assembler) TASM(Turbo Assembler)

Macro Processor

- Macro is the unit of specification of program generation through expansion.
- Macros are special code fragments that are defined once in the program and used by calling them from various places within the program.
- Macro processor is a program that copies stream of text from one place to another, making a systematic set of replacements as it does so.
- They are often embedded in other programs such as assemblers and compilers.
    - It is simply a notational convenience for the programmer to write a shorthand version of a program(module programming).
    - It represents a commonly used group of statements in the source program.
    - It is replaced by the **macro processor** with the corresponding group of source language statements. This operation is called “expanding the macro”

For example:

- Suppose it is necessary to save the contents of all registers before calling a subroutine.
- This requires a sequence of instructions.

- We can define and use a macro, SAVEREGS, to represent this sequence of instructions
  Before you can use a macro, you must *define* it explicitly with the `#define' directive. `#define' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

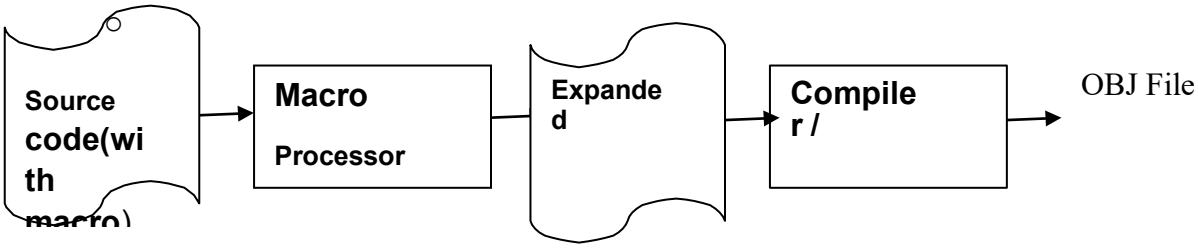
#define BUFFER_SIZE 1020

Defines a macro named `BUFFER_SIZE' as an abbreviation for the text `1020'

Its functions essentially involve the substitution of one group of characters or lines for another.

- Normally, it performs no analysis of the text it handles.
- It doesn't concern the meaning of the involved statements during macro expansion.
- Therefore, the design of a macro processor generally is machine independent.

- Macro processors are used in
  - assembly language
  - high-level programming languages, e.g., C or C++
  - OS command languages
  - general purpose



Format of macro definition

A macro can be defined as follows

MACRO Name [List of Parameters] -MACRO pseudo-op shows start of macro definition.

Macro name with a list of formal parameters.

….

……. Sequence of assembly language instructions.

MEND MEND (MACRO-END) Pseudo shows the end of macro definition.

Example:

MACRO SUM X, Y LDA X MOV BX, X LDA Y

ADD BX

MEND

1 BASIC MACROPROCESSOR FUNCTIONS

The fundamental functions common to all macro processors are:

1. Macro Definition
2. Macro Invocation
3. Macro Expansion

Macro Definition and Expansion

- Two new assembler directives are used in macro definition:

label op operands name
MACRO parameters

:

body

: MEND

  - MACRO: identify the beginning of a macro definition
  - MEND: identify the end of a macro definition
- Prototype for the macro: Each parameter begins with '&'
- Body: The statements that will be

generated as the expansion of the macro.

```
5       COPY    START   0               COPY FILE FROM INPUT TO OUTPUT
10      RDBUFF  MACRO   &INDEV,&BUFADR,&RECLTH
15      .
20      .       MACRO TO READ RECORD INTO BUFFER
25      .
30              CLEAR   X               CLEAR LOOP COUNTER
35              CLEAR   A
40              CLEAR   S
45              +LDT    #4096           SET MAXIMUM RECORD LENGTH
50              TD      =X'&INDEV'      TEST INPUT DEVICE
55              JEQ     *-3             LOOP UNTIL READY
60              RD      =X'&INDEV'      READ CHARACTER INTO REG A
65              COMPR   A,S             TEST FOR END OF RECORD
70              JEQ     *+11            EXIT LOOP IF EOR
75              STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
80              TIXR    T               LOOP UNLESS MAXIMUM LENGTH
85              JLT     *-19              HAS BEEN REACHED
90              STX     &RECLTH         SAVE RECORD LENGTH
95              MEND
100     WRBUFF  MACRO   &OUTDEV,&BUFADR,&RECLTH
105     .
110     .       MACRO TO WRITE RECORD FROM BUFFER
115     .
120             CLEAR   X               CLEAR LOOP COUNTER
125             LDT     &RECLTH
130             LDCH    &BUFADR,X       GET CHARACTER FROM BUFFER
135             TD      =X'&OUTDEV'     TEST OUTPUT DEVICE
140             JEQ     *-3             LOOP UNTIL READY
145             WD      =X'&OUTDEV'     WRITE CHARACTER
150             TIXR    T               LOOP UNTIL ALL CHARACTERS
155             JLT     *-14              HAVE BEEN WRITTEN
160             MEND
165     .
170     .       MAIN PROGRAM
175     .
180     FIRST   STL     RETADR          SAVE RETURN ADDRESS
190     CLOOP   RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
195             LDA     LENGTH          TEST FOR END OF FILE
200             COMP    #0
205             JEQ     ENDFIL          EXIT IF EOF FOUND
210             WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
215             J       CLOOP           LOOP
220     ENDFIL  WRBUFF  05,EOF,THREE    INSERT EOF MARKER
225             J       @RETADR
230     EOF     BYTE    C'EOF'
235     THREE   WORD    3
240     RETADR  RESW    1
245     LENGTH  RESW    1               LENGTH OF RECORD
250     BUFFER  RESB    4096            4096-BYTE BUFFER AREA
255             END     FIRST
```

It shows an example of a SIC/XE program using macro Instructions.

- This program defines and uses two macro instructions, RDBUFF and WRDUFF .
- Two Assembler directives (MACRO and MEND) are used in macro definitions.
- The first MACRO statement identifies the beginning of macro definition.
- The Symbol in the label field (RDBUFF) is the name of macro, and entries in the operand field identify the parameters of macro instruction.
- In our macro language, each parameter begins with character &, which facilitates the substitution of parameters during macro expansion.
- The macro name and parameters define the pattern or prototype for the macro instruction used by the programmer. The macro instruction definition has been deleted since they have been no longer needed after macros are expanded.
- Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from macro invocation substituted for the parameters in macro prototype.
- The arguments and parameters are associated with one another according to their positions.

Macro Invocation

- A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro.
- The processes of macro invocation and subroutine call are quite different.
    - Statements of the macro body are expanded each time the macro is invoked.
    - Statements of the subroutine appear only one; regardless of how many times the subroutine is called.
- The macro invocation statements treated as comments and the statements generated from macro expansion will be assembled as though they had been written by the programmer.

Macro Expansion

- Each macro invocation statement will be expanded into the statements that form the body of the macro.
- Arguments from the macro invocation are substituted for the parameters in the macro prototype.

- The arguments and parameters are associated with one another according to their positions.

- The first argument in the macro invocation corresponds to the first parameter in the macro prototype, etc.
- Comment lines within the macro body have been deleted, but comments on individual statements have been retained.
- Macro invocation statement itself has been included as a comment line.

Example of a macro expansion

```
  5     COPY     START   0                   COPY FILE FROM INPUT TO OUTPUT
180     FIRST    STL     RETADR              SAVE RETURN ADDRESS
190     .CLOOP   RDBUFF  F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
190a    CLOOP    CLEAR   X                   CLEAR LOOP COUNTER
190b             CLEAR   A
190c             CLEAR   S
190d             +LDT    #4096               SET MAXIMUM RECORD LENGTH
190e             TD      =X'F1'              TEST INPUT DEVICE
190f             JEQ     *-3                 LOOP UNTIL READY
190g             RD      =X'F1'              READ CHARACTER INTO REG A
190h             COMPR   A,S                 TEST FOR END OF RECORD
190i             JEQ     *+11                EXIT LOOP IF EOR
190j             STCH    BUFFER,X            STORE CHARACTER IN BUFFER
190k             TIXR    T                   LOOP UNLESS MAXIMUM LENGTH
190l             JLT     *-19                  HAS BEEN REACHED
190m             STX     LENGTH              SAVE RECORD LENGTH
```

- In expanding the macro invocation on line 190, the argument F1 is substituted for the parameter and INDEV wherever it occurs in the body of the macro.

- Similarly BUFFER is substituted for BUFADR and LENGTH is substituted for RECLTH.
- Lines 190a through 190m show the complete expansion of the macro invocation on line 190.
- The label on the macro invocation statement CLOOP has been retained as a label on the first statement generated in the macro expansion.
- This allows the programmer to use a macro instruction in exactly the same way as an assembler language mnemonic.
- After macro processing the expanded file can be used as input to assembler.
- The macro invocation statement will be treated as comments and the statements generated from the macro expansions will be assembled exactly as though they had been written directly by the programmer.

Linker

Role of linker is to combine multiple object files generated by compilers or assemblers into a single executable file.

- Process of collecting and combining various pieces of code and data in to single file that can be loaded in to memory and executed.
- Linking performed a compile time, when source code is translated to machine code, at load time, when program is loaded in to memory and executed by loader and at run time by application programs.
- Eg : GNU Linker, Microsoft linker(link.exe),GOLD(GNU Linker)

Types:

a) Linking Loader: Performs all linking and relocation operations directly in to main memory for execution.

b) Linkage Editor: Produce a linked version of program called as load module or executable image. This load module is written in to file or library for later execution.

c) Dynamic Linker: This linking postpones the linking function until execution time. Also called as dynamic loading.

Loader

Role of the loader is to load the executable file into memory and prepares it for execution, handover the control to the program.

Work flow – When the user types commands to run a program ,Operating system's loader takes over and start loading program. The loader reads the program code from the

disk(secondary storage) into memory(RAM).Loader relocates the addresses in the program code to reflect the actual memory locations. Loader also finds and locates the required libraries ,allocates memory spaces for stack (setup and initialize stack pointer and other registers)and heap and hand over the control for execution.

- Utility of an operating system.
- Copies program from a storage device to computer's main memory.
- They can replace virtual address with real address.
- They are invisible to user.

Linker vs Loader

Linker	Loader
Combines multiple object files into single one for execution	Loads executable files into memory and prepares for execution.
Operates during build process, before the program is executed	Operates at run time when user starts the program
Output- produces executable file that can run by operating System.	Output- produces a running program, makes the processor ready to execute
Requires compiled object files and libraries to create an executable file.	Requires executable file to load in memory for program execution.

Debugger

- An Interactive debugging system provides programmers with facilities that aid in testing and debugging of programs.It allows to execute code step by step and understand the flow of it to fix the bugs.
- Debugging means locating bugs or faults in program.
- Helps in fixing error.
- Determination of exact nature and location of error in the program.

- Debugger speeds up the process by identifying and fixing the bugs by allowing the programmer to inspect the program state at specific points and understand its work flow
- For Example ,

  We can check the variables and see each iteration of the FOR loop in a C code.

  Example for Debuggers : GDB for C++ , PDB for PYTHON,Builtin debuggers for VS,IntelliJ IDEA

Device Driver

- It is a software module which manages the communication and control of specific I/O device on type of device.
- Convert logical requests from the user in to specific commands directed to device itself.

Work flow- When the OS boots up or when a new hardware device is connected ,device driver is loaded into memory.Device drivers initialize the hardware device registers or allocating resources.Applications and OS send request to the driver to perform operations like reading or writing .Device drivers translate the high level request into low level codes that the hardware can understand.

Memory devices generate interrupts to signal completion or an operation to indicate an event.It also manages data transfer between hardware and system memory,handles the errors that may occur during communication with hardware.

When the system shuts down or the hardware device is removed,device drivers perform necessary steps like releasing the allocated resources and deregistering from the OS.

Eg : Printer drivers ,Bluetooth drivers , Graphic devices(NVIDIA),Storage drivers

Text Editors

- Program that allows the user to create the source program in the form of text in to the main memory.
- Creation, edition, deletion, updating of document or files can be done with the help of text editor.

Key Features include

- **Syntax Highlighting**(with color coding for different elements for readability)
- **Auto completion**(Suggesting completions or parts of the typed word or codes)

- **Search or Replace**
- **Line Numbering**
- **Extensibility**(Supports plug ins to add more useability)

Eg: IntelliJ IDEA ,Eclipse,NetBeans,VS code (For java)

VS Code,PyCharm,Atom,Emacs(For PYTHON)

VS Code,Vim,Notepad++ (For C,C++)

SIC(Simplified Instructional Computer)

It is a hypothetical computer used for education purposes to help students understand the fundamentals of computer architecture and arithmetic and logical operations.

It is similar to a typical microcomputer. It comes in two versions:

- The standard model
- XE version

SIC Machine Architecture:

Memory:

- It consists of bytes (8 bits) ,words (24 bits which are consecutive 3 bytes) addressed by the location of their lowest numbered byte.
- There are totally 32,768 bytes in memory.

Registers:

- There are 5 registers namely
- Accumulator (A): Used Accumulator is a special purpose register used for arithmetic operations
- Index Register(X) Stores and calculates addresses.
- Linkage Register (L): Linkage register stores the return address of the jump of subroutine instructions (JSUB).
- Program Counter (PC): Program counter contains the address of the current instructions being executed.

- Status Word (SW): Status word contains a variety of information including the condition code.

Data formats:

- Integers are stored as 24-bit binary numbers: 2's complement representation is used for negative values characters are stored using their 8 bit ASCII codes.
- They do not support floating – point data items.

Instruction formats:

All machine instructions are of 24-bits wide

Opcode	X (1)	Address

X- flag bit that is used to indicate indexed-addressing mode.

Addressing modes:

- Two types of addressing are available namely,
    1. Direct addressing mode
    2. Indexed addressing mode or indirect addressing mode

Mode	Indication	Target Address calculation
Direct	X=0	TA=Address
Indexed	X=1	TA=Address + (X)

- Where(x) represents the contents of the index register(x)

Instruction set:

It includes instructions like:

1. Data movement instruction: Basic set of instructions load and store registers. Ex: LDA, LDX, STA, STX.
2. Arithmetic operating instructions: Arithmetic operations involve register A and a word in memory and result is left in the register.

Ex: ADD, SUB, MUL, DIB.

3. Branching instructions Ex: JLT, JEQ, TGT.
4. Subroutine linkage instructions: These instructions are used for subroutine linkage. Ex: **JSUB**: jumps to the subroutine placing the return address in register L.

RSUB: returns by jumping to the address contained in register L

Input and Output:

I/O is performed by transferring one 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 3 I/O instructions,
  1) The Test Device (TD) instructions tests whether the addressed device is ready to send or receive a byte of data. Check condition code to see if device is ready. If CC setting is < the device is ready, if setting is = device is not ready.
  2) A program must wait until the device is ready, and then execute a Read Data (RD) or Write Data (WD).
  3) The sequence must be repeated for each byte of data to be read or written.

SIC/XE ARCHITECTURE & SYSTEM SPECIFICATION

Memory:

- 1 word = 24 bits (3 8-bit bytes)
- Total (SIC/XE) = 2^{20} (1,048,576) bytes (1Mbyte)

Registers:

- There are nine 24 bit registers

Mnemonic	Number	Special Use
A	0	Accumulator
X	1	Index register
L	2	Linkage register (JSUB/RSUB)
B	3	Base register
S	4	General register
T	5	General register
F	6	Floating Point Accumulator (48 bits)
PC	8	Program Counter (PC)
SW	9	Status Word (includes Condition Code, CC)

Data Format:

- Integers are stored in 24 bit, 2's complement format
- Characters are stored in 8-bit ASCII format
- Floating point is stored in 48 bit signed-exponent-fraction format:

s	exponent	fraction {36}

- The fraction is represented as a 36 bit number and has value between 0 and 1.
- The exponent is represented as a 11 bit unsigned binary number between 0 and 2047.
- The sign of the floating point number is indicated by s : 0=positive, 1=negative.
- Therefore, the absolute floating point number value is: $f*2^{(e-1024)}$

Instruction Format:

There are 4 different instruction formats available:

Format 1 (1 byte):

opcode {8}

Format 2 (2 bytes):

Opcode(8 bits)	Operand1(4bits)	Operend2(4 bits)

Format 3 (3 bytes):

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

n,i,x,b,p,e are flag bits of size 1 bit each.

Format 4 (4 bytes):

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

- Formats 3 & 4 introduce addressing mode flag bits:

- n=0 & i=1
- ***Immediate addressing*** - TA is used as an operand value (no memory reference)
- n=1 & i=0
- ***Indirect addressing*** - word at TA (in memory) is fetched & used as an address to fetch the operand from
- n=0 & i=0
- ***Simple addressing*** TA is the location of the operand
- n=1 & i=1
- ***Simple addressing*** same as n=0 & i=0 Flag x:
- x=1 Indexed addressing adds contents of X register to TA calculation Flag b & p (Format 3 only):
- b=0 & p=0

  Direct addressing displacement/address field contains TA (Format 4 always uses direct addressing)
- · ***b=0 & p=1***
- ***PC relative addressing*** - TA=(PC)+disp (-2048<=disp<=2047)
- b=1 & p=0
- ***Base relative addressing*** - TA=(B)+disp (0<=disp<=4095)

Flag e:

If e = 0, Use Format 3

If e=1,Use Format 4

Instructions set in SIC/XE

Data Movement Instructions

LDA (Load Accumulator)

LDX (Load Index Register)

LDL (Load Linkage Register)

STA (Store Accumulator)

STX (Store Index Register)

STL (Store Linkage Register)

LDT (Load T Register)

STT (Store T Register)

Arithmetic Instructions

ADD (Add to Accumulator)

SUB (Subtract from Accumulator)

MUL (Multiply Accumulator)

DIV (Divide Accumulator)

Comparison Instructions

COMP (Compare Accumulator)

COMPR (Compare Register)

CLEAR (Clear Register)

Branch Instructions

J (Jump)

JEQ (Jump if Equal)

JGT (Jump if Greater)

JLT (Jump if Less Than)

JSUB (Jump to Subroutine)

RSUB (Return from Subroutine)

Logical Instructions

AND (Logical AND)

OR (Logical OR)

TIX (Test and Increment Index)

Shift and Rotate Instructions

SHIFTL (Shift Left)

SHIFTR (Shift Right)

RMO (Register Move)

SVC (Supervisor Call)

I/O Instructions

- **Input and Output (I/O):**

- 2^8 (256) I/O devices may be attached, each has its own unique 8-bit address
- 1 byte of data will be transferred to/from the rightmost 8 bits of register A Three I/O instructions are provided:
- RD Read Data from I/O device into A
- WD Write data to I/O device from A
- TD Test Device determines if addressed I/O device is ready to send/receive a byte of data. The CC (Condition Code) gets set with results from this test:
- < *device is ready to send/receive*
- = *device isn't ready*
- SIC/XE Has capability for programmed I/O (I/O device may input/output data while CPU does other work) - 3 additional instructions are provided:
- SIO Start I/O
- HIO Halt I/O
- TIO Test I/O

Basic Structure of a SIC Program using assembler directives

START directive: Indicates the starting address of the program.

Instructions: Consist of operation codes (opcodes) and operands.

Variables/Constants: Defined using **WORD f**or constants and **RESW** for reserved words.

END directive: Marks the end of the program and specifies the starting point for execution.

Example 1: Simple Addition of two Numbers. This program adds two numbers and stores the result in memory.

```
START   1000
        LDA     NUM1            ;
Load NUM1 into the accumulator
        ADD     NUM2            ; Add
NUM2 to the value in the accumulator
        STA     RESULT          ;
Store the result in RESULT
        HLT                     ;
Halt the program

NUM1    WORD    5               ;
First number
NUM2    WORD    10              ;
Second number
RESULT  RESW    1               ;
Reserve one word for the result

        END     START           ; End
of the program
```