CST 305

SYSTEM SOFTWARE

CO#	Course Outcomes
CO1	Distinguish **softwares** into system and application software categories. **(Cognitive Knowledge Level: Understand)**
CO2	Identify **standard** and extended architectural features of machines. **(Cognitive Knowledge Level: Apply)**
CO3	Identify machine dependent features of system software **(Cognitive Knowledge Level: Apply)**
CO4	Identify machine independent features of system software. **(Cognitive Knowledge Level: Understand)**
CO5	Design **algorithms** for system softwares and analyze the effect of data structures. **(Cognitive Knowledge Level: Apply)**
CO6	Understand the features of device drivers and editing & debugging tools.**(Cognitive Knowledge Level: Understand)**

Mark Distribution

Total Marks	CIE Marks	ESE Marks	ESE Duration
150	50	100	3

Continuous Internal Evaluation Pattern:

Attendance : **10 marks**

Continuous Assessment Test (Average of series Tests 1&2) : **25 marks**

Continuous Assessment Assignment : **15 marks**

Text book

1. Leland L. Beck, System Software: An Introduction to Systems Programming, 3/E, Pearson Education Asia

Module-1
(Introduction)

Contents

- 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.

System Software

- system software consist of a variety of programs that support the operation of computer
- it make it possible for user to focus on appplication/problem to be solved without needing to know the details of how machine work internally

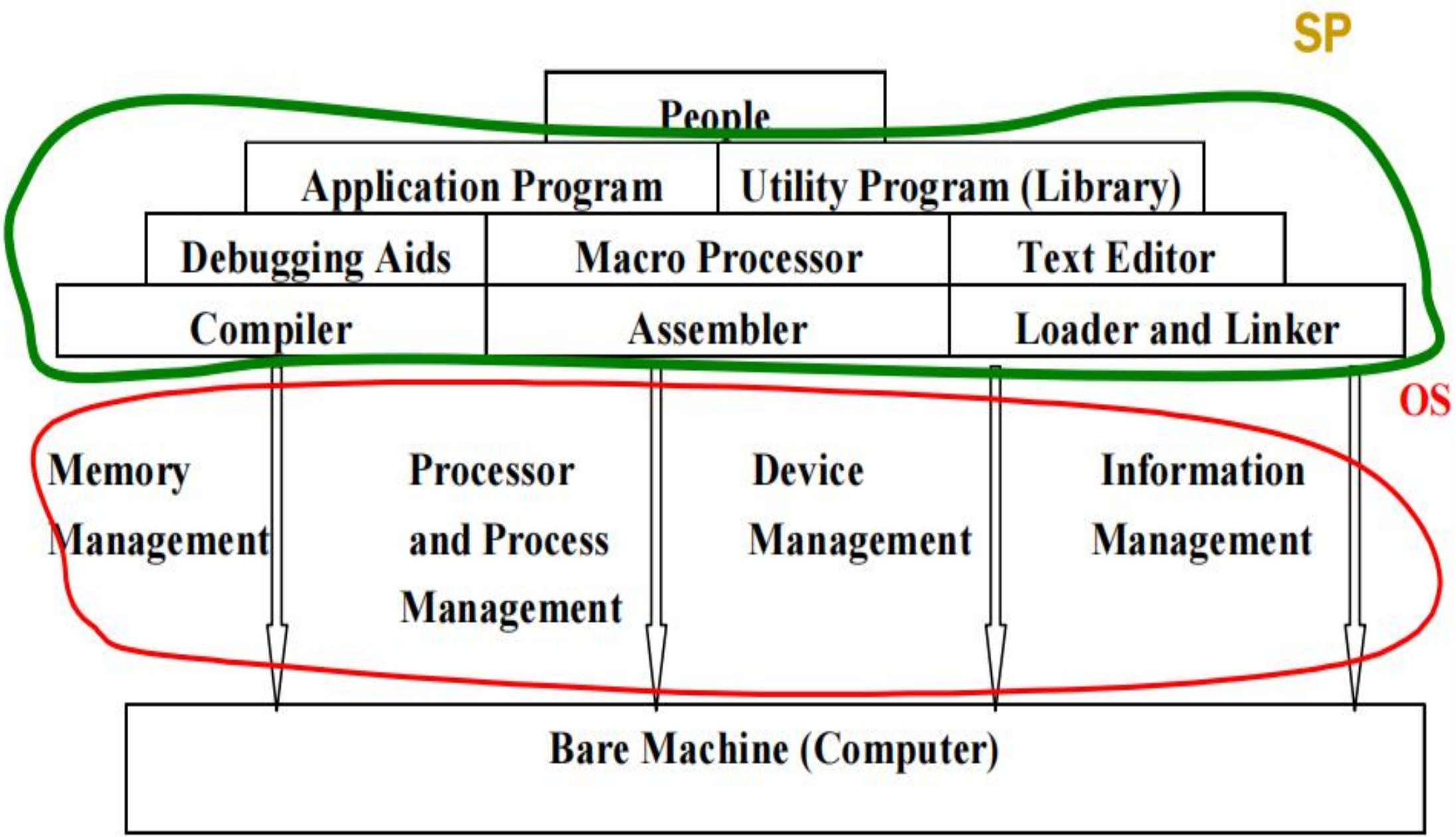
Application Software

- Application software (App) is a kind of software that performs specific functions for the end user by interacting directly with it.
- The sole purpose of application software is to aid the user in doing specified tasks.
- Web browsers like Firefox, Google Chrome, as well as Microsoft Word ,Excel are examples of application software

System Software vs Application Software

- **Machine dependency**-system software is related to the architecture of machine/machine dependency on which they are to run
- system program support operation and use of computer rather than any application
- Application program focus on application not on computing system and is concerned with solution of some problem using computer as a tool

- **examples of machine dependencies**
- assemblers translate mnemonic instructions into machine code taking into consideration the instruction formats,addressing modes etc.
- compilers must generate machine language code,taking into account such hardware characteristics as the number and type of registers and the machine instructions available.
- Operating systems are directly concerned with the management of nearly all of the resources of a computing system.


SP
People
Application Program
Utility Program (Library)
Debugging Aids
Macro Processor
Text Editor
Compiler
Assembler
Loader and Linker
OS
Memory Management
Processor and Process Management
Device Management
Information Management
Bare Machine (Computer)

Different System Software–

- Assembler
- Linker
- Loader
- Macro Processor
- Text Editor
- Debugger
- Device Driver
- Compiler
- Interpreter
- Operating System

System Software -Types

- Computers will only understand instructions and data that are in binary form (machine code), so any code written in a high-level language (Python, C++) will need to be translated into machine code in order to be executed by the CPU.
- There are three types of translator that can be used:
- compiler
- interpreter
- assembler

Assembler

- An assembler translates assembly language into machine code.
- Assembly language is a low-level language written in mnemonics that closely reflects the operations of the CPU.
- For example, a common mnemonic is STA (store data to memory).
- Each assembly language is particular to the computer/cpu and is not portable like high-level languages.
- Assembly code can directly access hardware and is often used for writing device drivers and embedded system.
- Assembly code is written directly for the CPU and hardware, and therefore code is very efficient as it does not need to be compiled or interpreted.

Compiler

- A compiler translates an entire program into machine code before execution.
- Discovering errors in the code can be difficult compared to interpreted languages, as all bugs are reported in an error report after the program has been compiled.
- An interpreter will discover errors as they come across them.
- Compilation is slow, but machine code can be executed quickly.
- Java and C++ are examples of compiled programming languages.

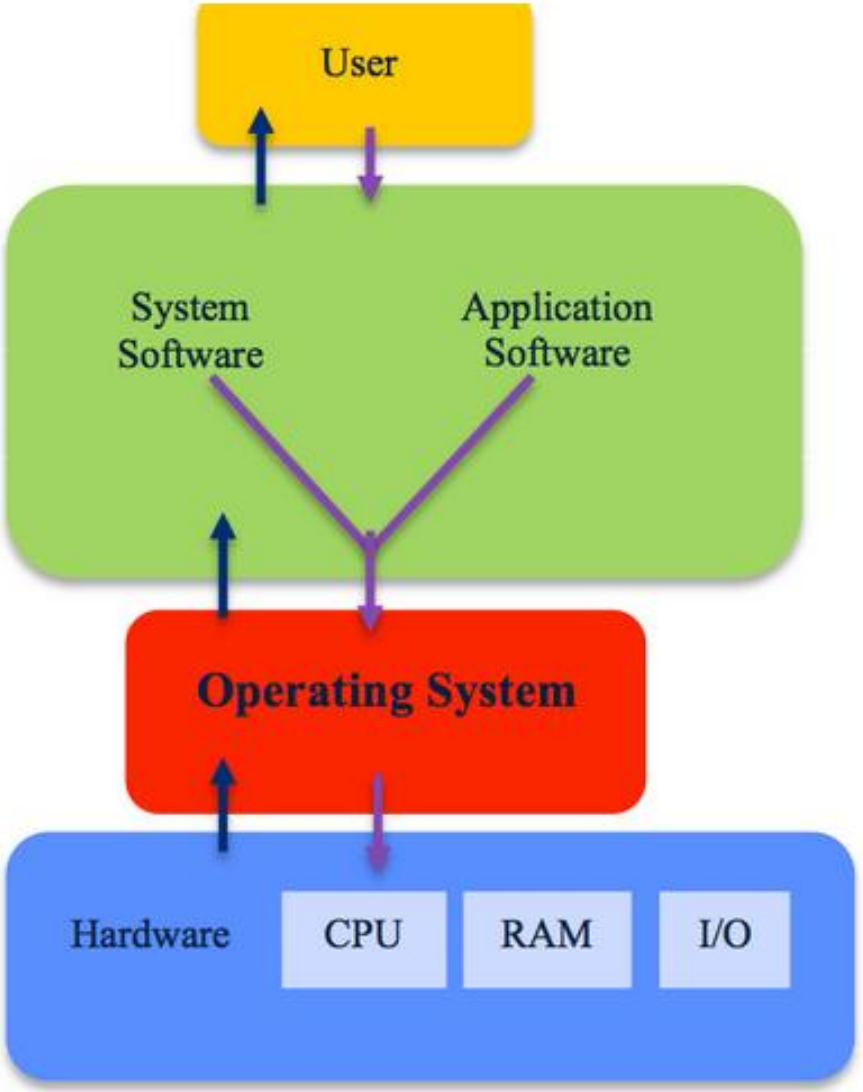
Interpreter

- Each time the program is run, an interpreter translates the code into machine code, instruction by instruction.
- The CPU executes each instruction before the interpreter moves on to translate the next instruction.
- Interpreted code will show an error as soon as it hits a problem, so it is easier to debug than compiled code.
- An interpreter does not create an independent final set of source code – source code is created each time it runs.
- Interpreted code is slower to execute than compiled code.
- Interpreter languages include JavaScript, PHP, Python and Ruby.

Operating System

- An operating system is a program that acts as an interface between the computer user and computer hardware, and controls the execution of programs.
- The operating system (OS) manages all of the software and hardware on the computer.
- It performs basic tasks such as file, memory and process management, handling input and output, and controlling peripheral devices such as disk drives and printers.
- Most of the time, there are several different computer programs running at the same time, and they all need to access your computer's central processing unit (CPU), memory and storage.
- The operating system coordinates all of this to make sure each program gets what it needs.
- most common operating systems are Microsoft's Windows, Apple's macOS and Linux

Operating System



Linker

- A linker is an important utility program that takes the object files, produced by the assembler and compiler, and other code to join them into a single executable file.
- The linker combines all external programs (such as libraries and other shared components) with our program to create a final executable for our program.
- So, the linker takes all object files as input, resolves all memory references, and finally merges these object files to make an executable file.
-

Loader

- The purpose of a linker is to produce executable files whereas the major aim of a loader is to load executable files to the memory.
- a loader is a vital component of an operating system for loading programs and libraries.
- The loader is a specialized operating system module that comes last and it loads the final executable code into memory.

Macro Processor

- A Macro represents a commonly used group of statements in the source programming language.
- A macro instruction (macro) allows the programmer to write shorthand version of a program (module programming)
- The macro processor replaces each macro instruction with the corresponding group of source language statements (expanding)
- o Normally, it performs no analysis of the text it handles.
- o It does not concern the meaning of the involved statements during macro expansion.
- The design of a macro processor generally is machine independent!
- Two new assembler directives are used in macro definition
- o MACRO: identify the beginning of a macro definition
- o MEND: identify the end of a macro definition
- **Prototype for the macro**
- o Each parameter begins with '&'
- name MACRO parameters
- :body //Body: the statements that will be generated as the expansion of the macro.
- :MEND

Text Editor

- A text editor is a program that allows us to create or edit programs and text files.
- Windows operating system has a text editor program called Notepad.
- Using Notepad, we can create a new program file, modify an existing file, or display or print the contents of a file.
- Is Microsoft Word a true text editor.?
- Most assemblers and compilers come with built-in text editors used to create a program and then assemble or compile it without having to exit from the editor.
- These editors also provide additional features, such as automatic key word highlighting, syntax checking, parenthesis matching, comment-line identification, and so on.

Debugger

- A debugger is a tool that allows you to observe a program as it runs, monitoring its state and watching the flow of execution.
- Most debuggers are capable of displaying the state of the CPU and its registers, memory, and various other aspects of a program at any point during execution.
- Most debuggers also allow you to set breakpoints that pause the program when memory at a specific address is executed, written, or read.
- Debuggers are another method of researching vulnerabilities within a program.
- Debuggers can be used to find problems within a program while it runs and to control the flow of a program as it executes.
- With a debugger, the whole of the program may be executed, or just certain parts.

Device Driver

- A device driver is a piece of software that enables communication between an operating system/application and hardware/peripheral devices.
- It serves as a bridge between the different components of a computer, allowing them to interact with each other.
- Without device drivers, computers would be unable to run any type of application because all programs require access to the underlying hardware in order to function.
- Device drivers enable the transmission of data from one component to another, simplifying the process by which users can interact with their machines

Device Driver

- Device drivers are broadly split into two categories:
- generic and specific.
- Generic device drivers are used for broad purposes such as keyboard control and mouse interaction.
- These are generally included as part of an operating system and can be used for basic functionality without any additional input from the user.
- Specific device drivers are made for individual pieces of hardware such as printers, scanners, audio cards, video cards and graphics cards.
- These usually need to be installed manually but may come packaged alongside certain devices or peripherals when purchased new.
- How do device drivers interact with hardware?

Device Driver

- Device drivers provide information about how software interacts with particular pieces of hardware.
- They communicate directly with the CPU on behalf of applications and user commands, sending instructions and retrieving data where necessary in order to execute a task.
- This provides two-way communication between the two components; data is sent from the hardware back up to applications so they can then interpret it appropriately (e.g., displaying content on screen).
- Similarly, data sent down from applications can then be translated by the driver into instructions which are understandable by the processor and executed on its behalf - allowing users to control their machines using software rather than low-level commands sent directly to the CPU itself.

- **Device drivers-**
- -Anatomy of a device driver
- -Character and block device drivers
- -General design of device drivers.
- **Text Editors-**
- -Overview of Editing
- -User Interface
- -Editor structure
- **Debuggers-**
- -Debugging functions and capabilities
- -Relation with other parts of the system
- -Debugging methods-Induction ,Deduction,Backtracking

- Write HLL programs using **text editor**-create and modify program
- Used **compiler** to translate these programs to machine language
- machine language program are then loaded into memory and ready for execution by **loader/Linker**
- **Debugger** detected errors in our program

- **assemblers** translate mnemonic instructions into machine code taking into account the instruction formats,addressing modes,etc
- **Operating systems** are directly concerned with the management of nearly all of the resources of a computing system and took care of all machine level details also so that user can concentrate on whatever they wanted to do without worrying about implementation.

- most system software is machine-dependent,real machines and software need to be considered
- However,most real computers have certain characteristics that are unusual or unique.
- It can be difficult to distinguish between those features of the software that are truly fundamental and those that depend solely on a particular machine.
- To avoid this problem,we present the fundamental functions of each piece of software through discussion of a Simplified Instructional Computer(SIC).

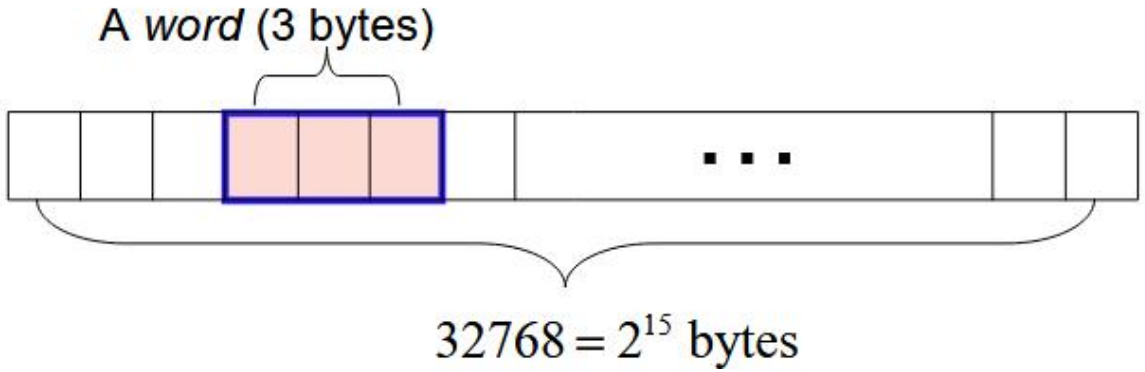
SIC

- Hypothetical computer designed to include commonly encountered hardware features and concepts most often found on real machines avoiding irrelavant complexities
- software concept can be separated from implementation details associated with that machine

- SIC come in 2 models :
- **Standard model**
- **XE version(**extra equipment /extra expensive**)**
- 2 versions are upward compatible means object program for standard SIC machine will also execute properly on SIC/XE system
- All SIC/XE machines have a special hardware feature designed to provide the upward compatability

SIC Machine Architecture

- **Memory**
- Memory consists of 8-bit bytes;
- any 3 consecutive bytes form a **word** (24 bits).
- All addresses on SIC are byte addresses;
- words are addressed by the location of their lowest numbered byte.
- There are a total of 32,768 (2^{15})bytes in the computer memory.



Registers

- There are five registers,all of which have special uses.
- Each register is 24 bits in length.

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

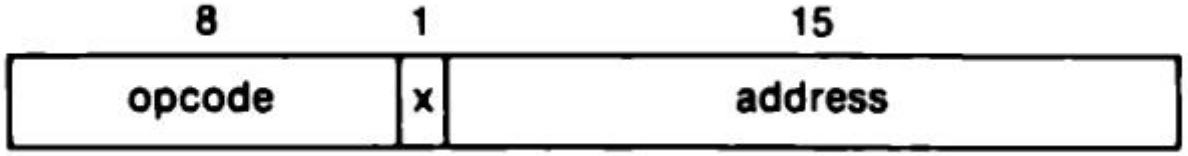
SIC does not have any stack.
It uses the linkage register to store the return address.
It is difficult to write the recursive program.
A programmer has to maintain memory for return addresses when we write more than one layer of function call.

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
- There is no floating-point hardware on the standard version of SIC.

instruction format

- All machine instructions on the standard version of SIC have the following 24-bit format:
- The flag bit x is used to indicate indexed-addressing mode.



Addressing Modes

- There are two addressing modes available,indicated by the setting of the x bit in the instruction.
- Parentheses are used to indicate the contents of a register or a memory location.
- For example,(X)represents the contents of register X.

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

target address is calculated from the address given in the instruction

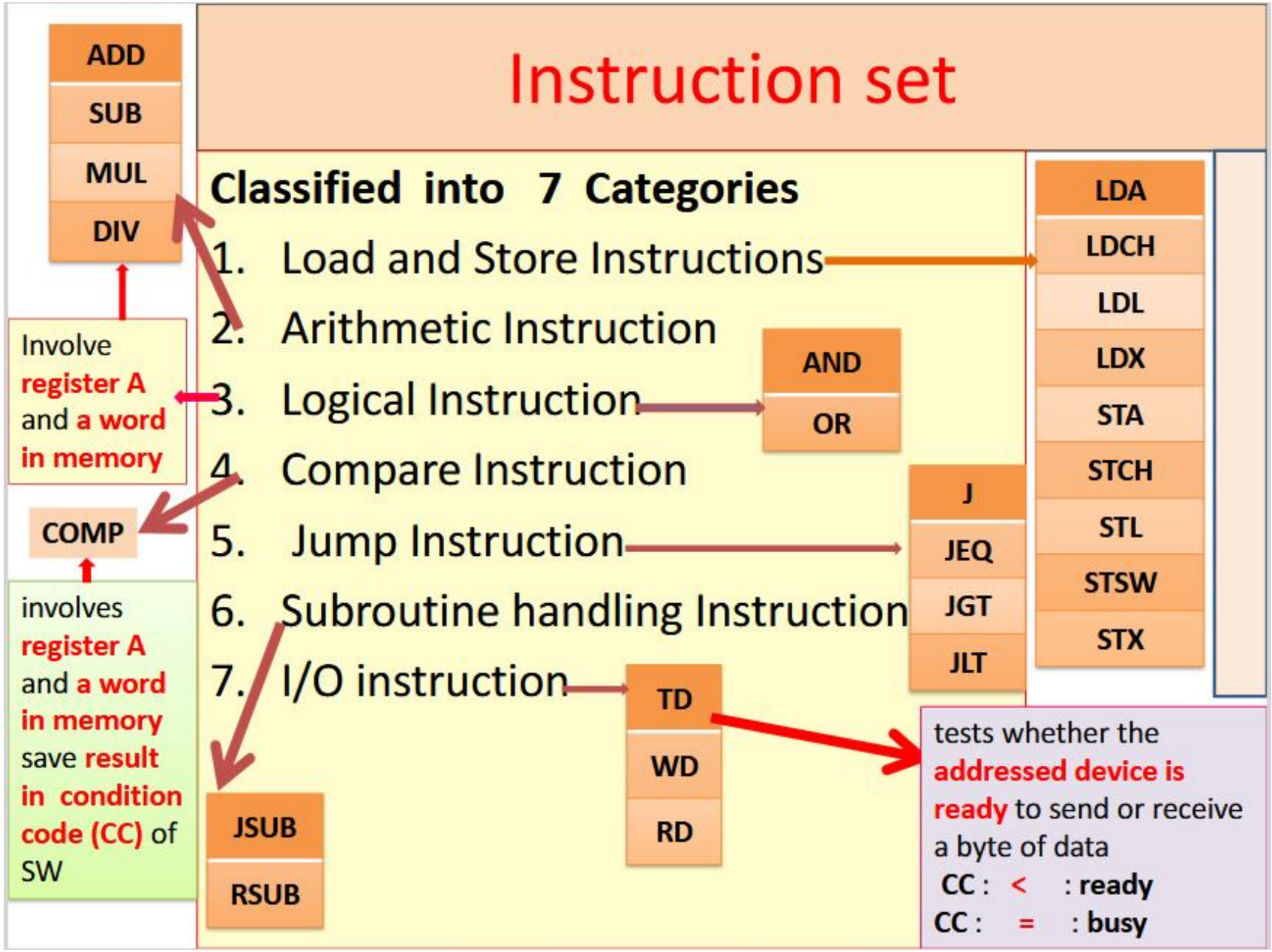
Instruction Set

- instructions that load and store registers (LDA,LDX,STA,STX,etc.)
- integer arithmetic operations (ADD,SUB,MUL,DIV).
- All arithmetic operations involve register A and a word in memory,with the result being left in the register.
- (COMP) instruction compares the value in register A with a word in memory;this instruction sets a condition code CC to indicate the result (<,=,or >).
- Conditional jump instructions (JLT,JEQ,JGT)can test the setting of CC and jump accordingly.
- Subroutine linkage instructions JSUB jumps to the subroutine placing the return address in register L
- RSUB returns by jumping to the address contained in register L.

Instruction set

Classified into 7 Categories

1. Load and Store Instructions
2. Arithmetic Instruction
3. Logical Instruction
4. Compare Instruction
5. Jump Instruction
6. Subroutine handling Instruction
7. I/O instruction



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.
- The Test Device(TD)instruction tests whether the addressed device is ready to send or receive a byte of data.
- The condition code is set to indicate the result of this test.
- (A setting of <means the device is ready to send or receive,and =means the device is not ready.)
- A program needing to transfer data must wait until the device is ready, then execute a Read Data (RD)or Write Data(WD).
- This sequence must be repeated for each byte of data to be read or written.

SIC/XE Machine Architecture

- ***Memory***
- The memory structure for SIC/XE is the same as SIC.
- Maximum memory on a SIC/XE is 1 Mbyte(2^{20} bytes)
- Addressing mode and instruction format changes accordingly

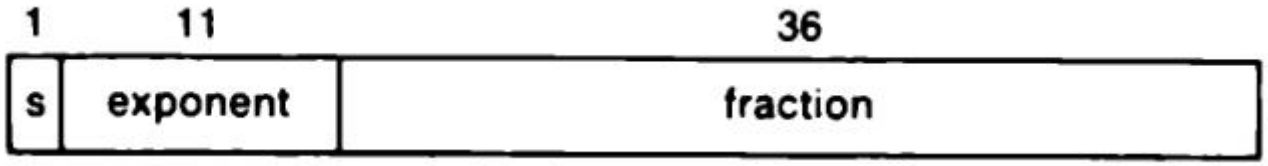
Registers

- Additional registers are provided by SIC/XE

Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register—no special use
T	5	General working register—no special use
F	6	Floating-point accumulator (48 bits)

Data Formats

- SIC/XE provides the same data formats as SIC.
- In addition, a 48-bit floating-point data type is also provided.

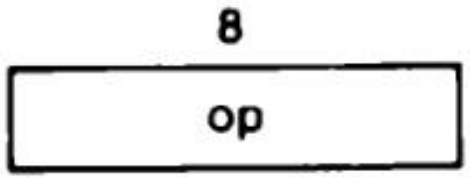


- Sign of the floating point number is indicated by value of s (0=positive ,1=negative)
- [s=e=f=0 to represent zero)
- Fraction is interpreted as value between 0 and 1
- Exponent is unsigned binary number between 0 and 2047
- If the exponent has value e and fraction has value f the absolute value of number represented is $f*2^{(e-1024)}$

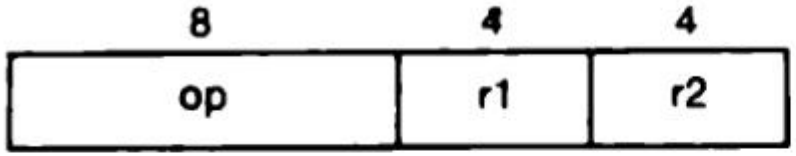
Instruction Formats

- Larger memory in SIC/XE means that address will no longer fit into 15 bit field so SIC instruction format cannot be used
- Available Options:
- use relative addressing(format 3)
- extend address field to 20 bits(format 4)
- instruction that do not reference memory at all (Format 1 and format 2)

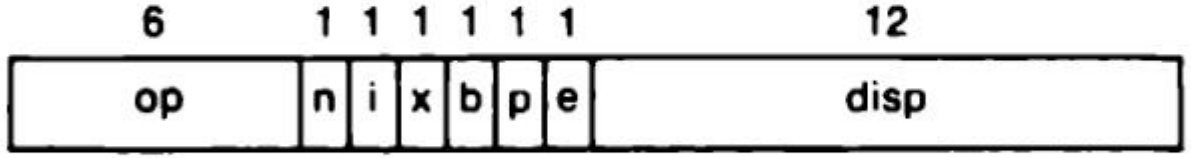
Format 1 (1 byte):



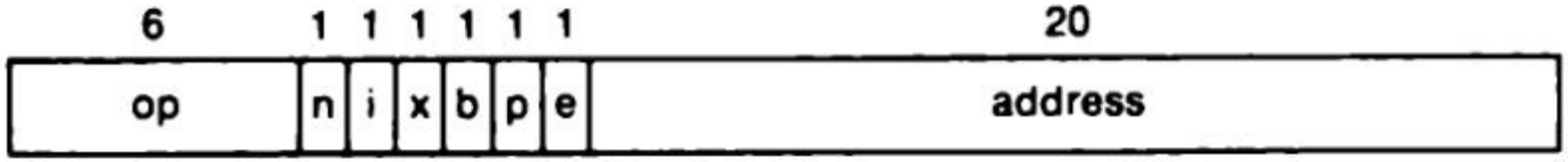
Format 2 (2 bytes):



Format 3 (3 bytes):



Format 4 (4 bytes):

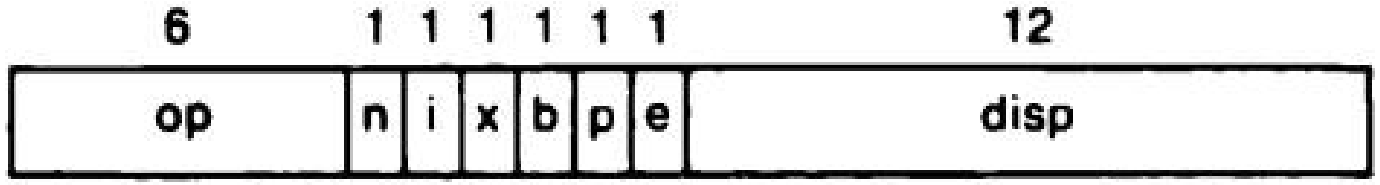


Addressing Modes

- Two new ***relative addressing modes*** (**Base relative, Program-counter relative**) are available for use with instructions assembled using Format 3.

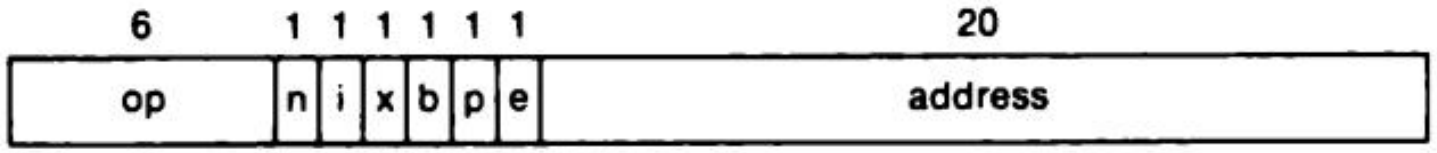
Mode	Indication	Target address calculation	
Base relative	b = 1, p = 0	TA = (B) + disp	(0 ≤ disp ≤ 4095)
Program-counter relative	b = 0, p = 1	TA = (PC) + disp	(–2048 ≤ disp ≤ 2047)

Format 3 (3 bytes):



- For base relative addressing,the displacement field **disp** in Format 3 instruction is interpreted as a 12-bit unsigned integer.
- For program-counter relative addressing,this field is interpreted as a 12-bit signed integer,with negative values represented in 2's complement notation.
- If bits b and p are both set to 0,the disp field from the Format 3 instruction is taken to be the target address.

Format 4 (4 bytes):



- For a Format 4 instruction,bits b and p are normally set to 0,and the target address is taken from the address field of the instruction.
- call this **direct addressing**,to distinguish it from the **relative** addressing modes
- Bit **e** is used to distinguish between format 3 and format 4
- **e=0 means Format 3**
- **e=1 means Format 4**

- Any of these addressing modes can also be combined with **indexed** addressing—if bit **x** is set to 1,the term(X)is added in the target address calculation.
- standard version of the SIC machine uses only **direct** addressing (with or without indexing).

- Bits i and n in Formats 3 and 4 are used to specify how the target address is used.
- If bit **i=1** and **n=0**,the target address itself is used as the operand value;no memory reference is performed.This is called **immediate addressing.**
- If bit **i=0** and **n=1**,the word at the location given by the target address is fetched;the value contained in this word is then taken as the address of the operand value.This is called **indirect addressing.**
- If bits **i** and **n** are both 0 or both 1,the target address is taken as the location of the operand;-**simple addressing.**
- indexing cannot be used with immediate or indirect addressing modes.
- SIC/XE instructions that specify **neither immediate nor indirect** addressing are assembled with bits **n and i both set to 1**
- Assemblers for the standard version of SIC will,set the bits in both of these positions to 0.(This is because the 8-bit binary codes for all of the SIC instructions end in 00.)

Addressing modes

- n i x b p e
- Simple n=0, i=0 (SIC) or n=1, i=1, TA=disp
- Immediate n=0, i=1 Disp=Value
- Indirect n=1, i=0 TA=(Operand)=(TA_1)
- Base relative b=1, p=0 TA=(B)+disp

  0 <= disp <= 4095
- PC relative b=0, p=1 TA=(PC)+disp

  -2048 <= disp <= 2047

Addressing Modes

Base relative addressing **- format 3 only**

- n = 1, i = 1, ***b=1, p=0***

Program-counter relative addressing **- format 3 only**

- n = 1, i = 1, ***b=0, p=1***

Direct addressing **– format 3 and 4**

- n = 1, i = 1, ***b=0, p=0***

Indexed addressing **– format 3 and 4**

- n = 1, i = 1, ***x = 1*** or n = 0, i = 0, ***x = 1***

Immediate addressing **– format 3 and 4**

- **n = 0, i = 1,** x = 0 // cannot combine with *indexed*

Indirect addressing **– format 3 and 4**

- **n = 1, i = 0,** x = 0 // cannot combine with *indexed*

Simple addressing **– format 3 and 4**

- **n = 0, i = 0 or n = 1, i = 1**

b	p	Addressing Mode	Target Address
0	0	Direct	TA = Disp + (X)
0	1	PC Relative	TA = (PC) + Disp + (X)
1	0	Base Relative	TA = (B) + Disp + (X)
1	1	-----	

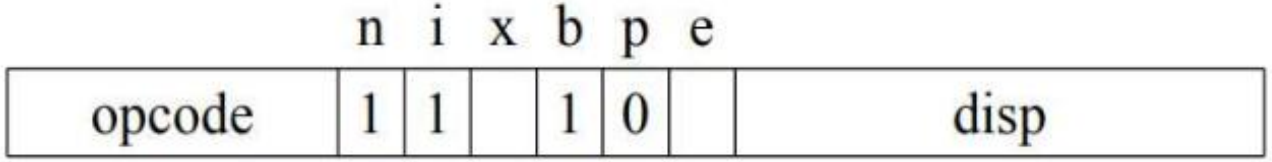
n	i	Addressing Mode	Operand
0	0	Simple Addressing	Operand = Contents of TA (bits b,p,e are part of address bits)
0	1	Immediate	Operand = TA
1	0	Indirect Addressing	Word at TA is the address of the Operand
1	1	Simple addressing	Operand = Contents of TA

Addressing mode

Direct	b=0, p=0	TA=disp
Index	x=1	$TA_{new}=TA_{old}+(X)$
Index+Base relative	x=1, b=1, p=0	TA=(B)+disp+(X)
Index+PC relative	x=1, b=0, p=1	TA=(PC)+disp+(X)
Index+Direct	x=1, b=0, p=0	TA=disp+(X)
Format 4	e=1	

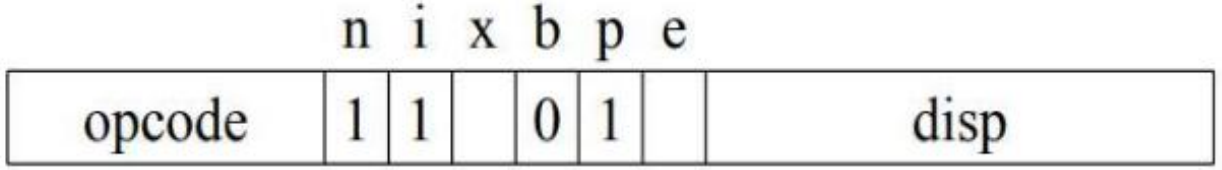
Addressing Modes: Address Computation

- ***Base Relative Addressing***



n=1, i=1, ***b=1, p=0***, TA=(**B**)+disp (0≤disp ≤4095)

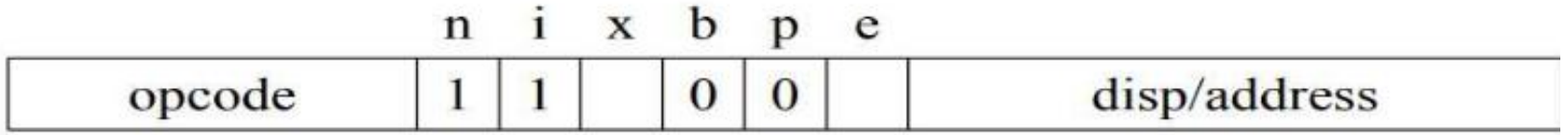
- ***Program-Counter Relative Addressing***



n=1, i=1, ***b=0, p=1***, TA=(**PC**)+disp (-2048≤disp ≤2047)

Addressing Modes: Address Computation

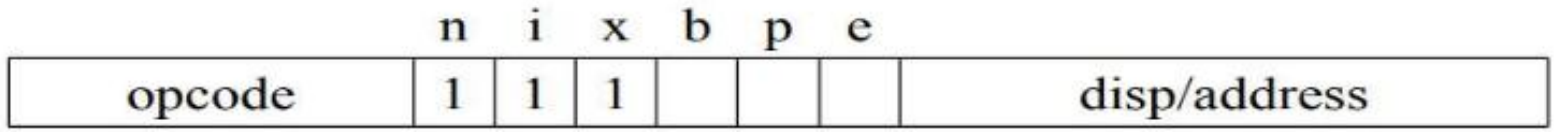
- ***Direct Addressing***
  - The target address is taken directly from the *disp* or *address* field



Format 3 (e=0): n=1, i=1, ***b=0, p=0***, TA=disp (0≤disp ≤4095)

Format 4 (e=1): n=1, i=1, ***b=0, p=0***, TA=address

- ***Indexed Addressing***
  - The term (X) is added into the target address calculation



n=1, i=1, ***x=1***

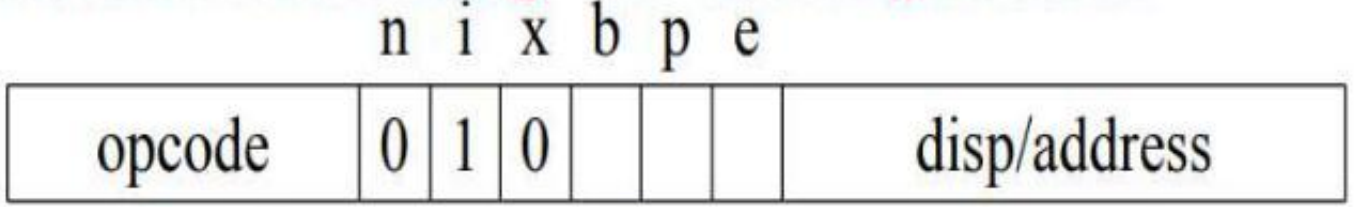
Ex. *Direct Indexed Addressing*

Format 3, TA=(X)+disp

Format 4, TA=(X)+address

Addressing Modes: Address Computation

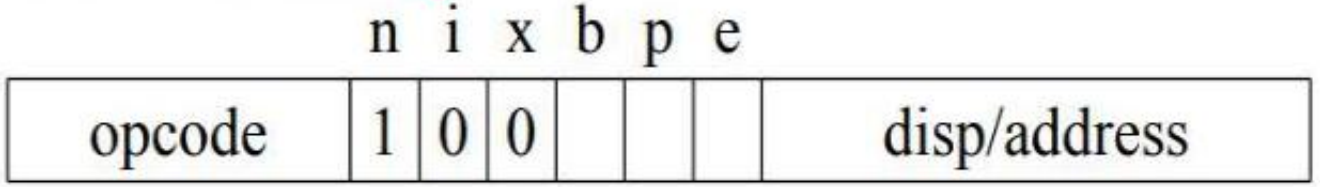
- ***Immediate Addressing – no memory access***



n=0, i=1, x=0, **operand**=disp //format 3

n=0, i=1, x=0, **operand**=address //format 4

- ***Indirect Addressing***

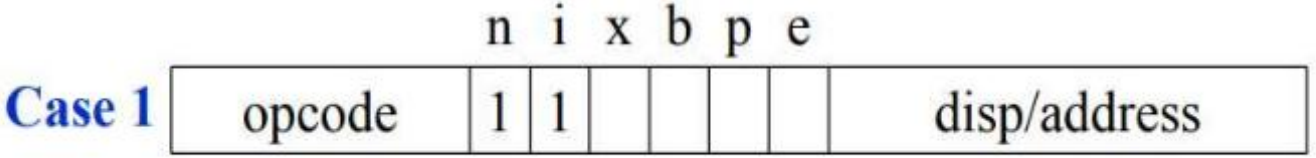


n=1, i=0, x=0, TA=(disp), **operand** = (TA) = ((disp))

n=1, i=0, x=0, TA=(address), **operand** = (TA) = ((address))

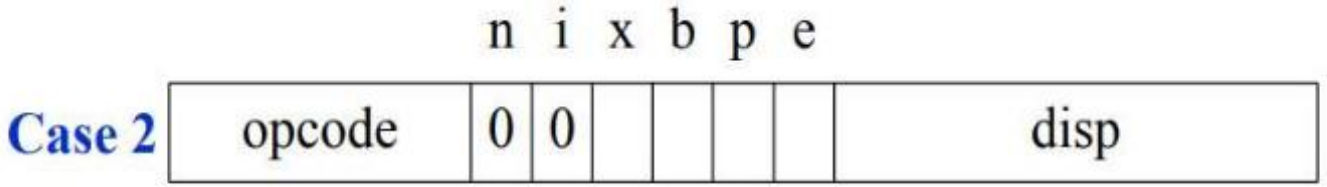
Addressing Modes: Address Computation

- ***Simple Addressing Mode***



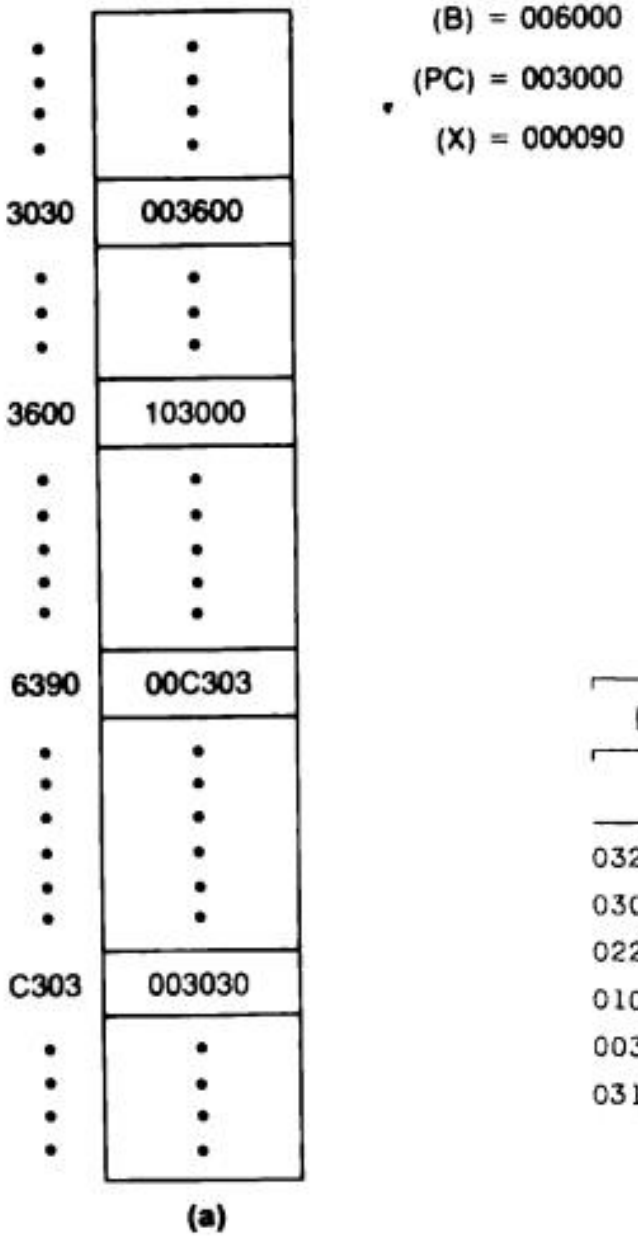
Format 3: ***i=1, n=1***, TA=disp, **operand** = (disp)

Format 4: ***i=1, n=1***, TA=address, **operand** = (address)



i=0, n=0, TA=b/p/e/disp (SIC standard)

- If bits n and i are both 0,then bits b,p,and e are considered to be part of the address field of the instruction
- special case of simple addressing
- neither immediate/indirect
- This makes Instruction Format 3 identical to the format used on the standard version of SIC,providing the desired compatibility.



-different addressing modes available on SIC/XE.
-contents of registers B,PC,and X,and of selected memory locations.(hexadecimal.)
-(b) machine code for a series of LDA instructions.
-The target address generated by each instruction,and the value that is loaded into register A

Machine instruction									
Hex	Binary							Target address	Value loaded into register A
	op	n	i	x	b	p	e disp/address		
032600	000000	1	1	0	0	1	0 0110 0000 0000	3600	103000
03C300	000000	1	1	1	1	0	0 0011 0000 0000	6390	00C303
022030	000000	1	0	0	0	1	0 0000 0011 0000	3030	103000
010030	000000	0	1	0	0	0	0 0000 0011 0000	30	000030
003600	000000	0	0	0	0	1	1 0110 0000 0000	3600	103000
0310C303	000000	1	1	0	0	0	1 0000 1100 0011 0000 0011	C303	003030

(b)

Instruction set

- provides all instructions available on standard SIC
- Additional instructions to load and store new registers(LDB,STB..)
- Instructions to perform floating point arithmetic operations(ADDF,SUBF,MULF,DIVF)
- instructions to take operands from registers(RMO)-Register Move instructions
- Register-Register arithmetic operations-ADDR,SUBR,MULR,DIVR
- Supervisor call instruction(SVC) generates an interupt for communication with OS

input and output

- SIC instructions are also available here as well
- In addition,there are I/O channels that can be used to perform input and output while the CPU is executing other instructions.
- This allows overlap of computing and I/O,resulting in more efficient system operation.
- The instructions SIO,TIO,and HIO are used to start,test,and halt the operation of I/O channels.

Programs

- examples of data movement operations for SIC and SIC/XE.
- There are no memory-to-memory move instructions,all data movement must be done using registers.

examples of data movement.

- 3-byte word is moved by loading it into register A and then storing the register at the desired destination.
- you can also use register X (instructions LDX,STX)or register L(LDL,STL).
- In the second example,a single byte of data is moved using the instructions LDCH(Load Character)and STCH(Store Character).
- These instructions operate by loading or storing the rightmost 8-bit byte of register A;the other bits in register A are not affected.

```
          LDA     FIVE        LOAD CONSTANT 5 INTO REGISTER A
          STA     ALPHA       STORE IN ALPHA
          LDCH    CHARZ       LOAD CHARACTER 'Z' INTO REGISTER A
          STCH    C1          STORE IN CHARACTER VARIABLE C1
           .
           .
           .
ALPHA     RESW    1           ONE-WORD VARIABLE
FIVE      WORD    5           ONE-WORD CONSTANT
CHARZ     BYTE    C'Z'        ONE-BYTE CONSTANT
C1        RESB    1           ONE-BYTE VARIABLE
```

- four different ways of **defining storage for data items** in the SIC assembler language.
- The statement WORD reserves one word of storage,which is initialized to a value defined in the operand field of the statement.
- Thus the WORD statement defines a data word labeled FIVE whose value is initialized to 5.
- The statement RESW reserves one or more words of storage for use by the program.
- For example,the RESW statement defines one word of storage labeled ALPHA,which will be used to hold a value generated by the program.
- The statements BYTE and RESB perform similar storage definition functions for data items that are characters instead of words.
- CHARZ is a 1-byte data item whose value is initialized to the character “Z”
- Cl is a 1-byte variable with no initial value.

- The data-movement operations instructions would also work on SIC/XE without advantage of the more advanced hardware features
- In this example,the value 5 is loaded into register A using immediate addressing.
- The operand field for this instruction contains the flag #(which specifies immediate addressing)and the data value to be loaded.
- character “Z”is placed into register A by using immediate addressing to load the value 90,which is the decimal value of the ASCII code that is used internally to represent the character “Z”.

```
        LDA     #5              LOAD VALUE 5 INTO REGISTER A
        STA      ALPHA          STORE IN ALPHA
        LDA     #90             LOAD ASCII CODE FOR 'Z' INTO REG A
        STCH     C1             STORE IN CHARACTER VARIABLE C1
          .
          .
          .
ALPHA   RESW     1              ONE-WORD VARIABLE
C1      RESB     1              ONE-BYTE VARIABLE
```

- examples of arithmetic instructions for SIC.
- All arithmetic operations are performed using register A,with the result being left in register A.
- sequence of instructions stores the value (ALPHA+INCR-1)in BETA and the value (GAMMA+INCR-1)in DELTA.

```
        LDA     ALPHA       LOAD ALPHA INTO REGISTER A
        ADD     INCR        ADD THE VALUE OF INCR
        SUB     ONE         SUBTRACT 1
        STA     BETA        STORE IN BETA
        LDA     GAMMA       LOAD GAMMA INTO REGISTER A
        ADD     INCR        ADD THE VALUE OF INCR
        SUB     ONE         SUBTRACT 1
        STA     DELTA       STORE IN DELTA
          .
          .
          .
ONE     WORD    1           ONE-WORD CONSTANT
.                           ONE-WORD VARIABLES
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

- calculations performed on SIC/XE.
- The value of INCR is loaded into register S initially,and the register-to-register instruction ADDR is used to add this value to register A when it is needed.
- This avoids having to fetch INCR from memory each time it is used in a calculation,which may make the program more efficient.
- Immediate addressing is used for the constant 1 in the subtraction operations.

```
        LDS     INCR        LOAD VALUE OF INCR INTO REGISTER S
        LDA     ALPHA       LOAD ALPHA INTO REGISTER A
        ADDR    S,A         ADD THE VALUE OF INCR
        SUB     #1          SUBTRACT 1
        STA     BETA        STORE IN BETA
        LDA     GAMMA       LOAD GAMMA INTO REGISTER A
        ADDR    S,A         ADD THE VALUE OF INCR
        SUB     #1          SUBTRACT 1
        STA     DELTA       STORE IN DELTA
        .
        .
        .
.                           ONE WORD VARIABLES
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

Looping and indexing operations

- loop that copies one 11-byte character string to another.
- The index register (register X)is initialized to zero before the loop begins
- during the first execution of the loop,the target address for the LDCH instruction will be the address of the first byte of STR1.
- the STCH instruction will store the character being copied into the first byte of STR2
- The next instruction,TIX,performs two functions.
- First it adds 1 to the value in register X,andthen it compares the new value of register X to the value of the operand (in this case,the constant value 11).
- The condition code is set to indicate the result of this comparison.
- The JLT instruction jumps if the condition code is set to“less than.”
- Thus,the JLT causes a jump back to the beginning of the loop if the new value in register X is less than 11.

- During the second execution of the loop,register X will contain the value1.
- Thus,the target address for the LDCH instruction will be the second byte of STR1,and the target address for the STCH instruction will be the second byte of STR2.
- The TIX instruction will again add 1 to the value in register X,and the loop will continue in this way until all 11 bytes have been copied from STR1 to STR2.
- Notice that after the TIX instruction is executed,the value in register X is equal to the number of bytes that have already been copied.

```
        LDX     ZERO            INITIALIZE INDEX REGISTER TO 0
MOVECH  LDCH    STR1,X          LOAD CHARACTER FROM STR1 INTO REG A
        STCH    STR2,X          STORE CHARACTER INTO STR2
        TIX     ELEVEN          ADD 1 TO INDEX, COMPARE RESULT TO 11
        JLT     MOVECH          LOOP IF INDEX IS LESS THAN 11
        .
        .
        .
STR1    BYTE    C'TEST STRING'  11-BYTE STRING CONSTANT
STR2    RESB    11              11-BYTE VARIABLE
.                               ONE-WORD CONSTANTS
ZERO    WORD    0
ELEVEN  WORD    11
```

SIC/XE

```
            LDT     #11             INITIALIZE REGISTER T TO 11
            LDX     #0              INITIALIZE INDEX REGISTER TO 0
MOVECH      LDCH    STR1,X          LOAD CHARACTER FROM STR1 INTO REG A
            STCH    STR2,X          STORE CHARACTER INTO STR2
            TIXR    T               ADD 1 TO INDEX, COMPARE RESULT TO 11
            JLT     MOVECH          LOOP IF INDEX IS LESS THAN 11
            .
            .
            .
STR1        BYTE    C'TEST STRING'  11-BYTE STRING CONSTANT
STR2        RESB    11              11-BYTE VARIABLE
```

Input output on sic

- example of input and output on SIC and the same instructions would also work on SIC/XE.
- This program fragment reads 1 byte of data from device F1 and copies it to device 05.
- The actual input of data is performed using the RD(Read Data)instruction.
- The operand for the RD is a byte in memory that contains the hexadecimal code for the input device (in this case,F1).
- Executing the RD instruction transfers 1 byte of data from this device into the rightmost byte of register A.
- If the input device is character-oriented (for example,a keyboard),the value placed in register A is the ASCII code for the character that was read.
- Before the RD can be executed,however,the input device must be ready to transmit the data.
- For example,if the input device is a keyboard,the operator must have typed a character.
- The program checks for this by using the TD(Test Device)instruction.
- When the TD is executed,the status of the addressed device is tested and the condition code is set to indicate the result of this test.
- If the device is ready to transmit data,the condition code is set to "less than";if the device is not ready,the condition code is set to “equal.”

Input output on sic

- program must execute the TD instruction and then check the condition code by using a conditional jump.
- if the condition code is "equal"(device not ready),the program jumps back to the TD instruction.
- This two-instruction loop will continue until the device becomes ready;then the RD will be executed.
- Output is performed in the same way.
- First the program uses TD to check whether the output device is ready to receive a byte of data.
- Then the byte to be written is loaded into the rightmost byte of register A,and the WD (WriteData)instruction is used to transmit it to the device.

```
INLOOP    TD      INDEV     TEST INPUT DEVICE
          JEQ     INLOOP    LOOP UNTIL DEVICE IS READY
          RD      INDEV     READ ONE BYTE INTO REGISTER A
          STCH    DATA      STORE BYTE THAT WAS READ
          .
          .
          .
OUTLP     TD      OUTDEV    TEST OUTPUT DEVICE
          JEQ     OUTLP     LOOP UNTIL DEVICE IS READY
          LDCH    DATA      LOAD DATA BYTE INTO REGISTER A
          WD      OUTDEV    WRITE ONE BYTE TO OUTPUT DEVICE
          .
          .
          .
INDEV     BYTE    X'F1'     INPUT DEVICE NUMBER
OUTDEV    BYTE    X'05'     OUTPUT DEVICE NUMBER
DATA      RESB    1         ONE-BYTE VARIABLE
```

- Variables ALPHA, BETA,GAMMA are arrays of 100 words each.Illustrate looping by adding together the corresponding elements of ALPHA and BETA and store result in GAMMA
- Read 100 byte record from input device to memory

SIC & SIC/XE Instruction set

INSTRUCTION SET TABLE

- *Uppercase letters refer to specific registers.*
- ***M** indicates a memory address*
- ***n** indicates an integer between 1 and 16*
- ***r1** and **r2** represent register identifiers.*

- *Parentheses are used to indicate the contents of a register or memory location.*

- ***A** ⟵**(m..m+2)** specifies that the contents of the memory location **m** through **m+2** are loaded into register A*

- ***m..m+2*** ***(A)*** *contents of register* ***A*** *are stored in the word that begins at address* ***m***.

- P Privileged Instruction
- X Instruction available only on XE version
- F Floating point Instruction
- C Condition code CC set to indicate result of operation

SIC/XE INSTRUCTION SET

Mnemonic	Format	Code	Effect	Notes
ADD **m**	3/4	18	A ← (A) + (m…m+2)	
ADDF	3/4	58	F ← (F) + (m…m+5)	X, F
ADDR **r1, r2**	2	90	r2 ← (r2) + (r1)	X
AND **m**	3/4	40	A ← (A) & (m…m+2)	
CLEAR **r1**	2	B4	r1 ← 0	X
COMP **m**	3/4	28	(A) : (m…m+2)	C
COMPF **m**	3/4	88	(F) : (m…m+5)	X, F, C
COMPR **r1, r2**	2	A0	(r1) : (r2)	X, C
DIV **m**	3/4	24	A ← (A) / (m…m+2)	
DIVF	3/4	64	F ← (F) / (m…m+5)	
DIVR **r1, r2**	2	9C	r2 ← (r2) / (r1)	X

SIC/XE INSTRUCTION SET

Mnemonic	Format	Code	Effect	Notes
FIX	1	C4	A ← (F) (Convert to Integer)	X, F
FLOAT	1	C0	F ← (A)(Convert to float)	X, F
J m	3/4	3C	PC ← m	
JEQ m	3/4	30	PC ← m if CC is set to =	
JGT m	3/4	34	PC ← m if CC is set to >	
JLT m	3/4	38	PC ← m if CC is set to <	
JSUB m	3/4	48	L ← (PC) ; PC ← m	
LDA m	3/4	00	A ← (m…m+2)	
LDB m	3/4	68	B ← (m…m+2)	X
LDCH m	3/4	50	A (rightmost byte) ← m	
LDF m	3/4	70	F ← (m…m+5)	X, F

SIC/XE INSTRUCTION SET

Mnemonic	Format	Code	Effect	Notes
LDL m	3/4	08	L ← (m…m+2)	
LDS m	3/4	6C	S ← (m…m+2)	X
LDT m	3/4	74	T ← (m…m+2)	X
LDX m	3/4	04	X ← (m…m+2)	
MUL m	3/4	20	A ← (A) * (m…m+2)	
MULF m	3/4	20	A ← (A) * (m…m+5)	X,F
MULR r1, r2	2	98	r2 ← (r2) * (r1)	
LDA m	3/4	00	A ← (m…m+2)	
NORM	1	C8	F ← (F) (normalized)	X, F
OR m	3/4	44	A ← (A) | (m…m+2)	X,F
RD m	3/4	D8	A (rightmost byte) ← Data	P

SIC/XE INSTRUCTION SET

Mnemonic	Format	Code	Effect	Notes
RMO r1 , r2	2	AC	r2 ← (r1)	X
RSUB	3/4	4C	PC ← (L)	X
SHIFTL r1, n	2	A4	r1 (r1) left shift n bits	X
SHIFTR r1, n	2	A8	r1 ← (r1) right shift n bits	X
STA **m**	3/4	0C	m…m+2 ← (A)	
STB **m**	3/4	78	m…m+2 ← (B)	X
STCH **m**	3/4	54	M ← (A) rightmost byte	
STF **m**	3/4	80	m…m+5 ← (F)	X, F
STL **m**	3/4	14	m…m+2 ← (L)	
STS **m**	3/4	7C	m…m+2 ← (S)	X
STT **m**	3/4	84	m…m+2 ← (T)	X

SIC/XE INSTRUCTION SET

Mnemonic	Format	Code	Effect	Notes
STX m	¾	10	m…m+2 ← (X)	
SUB m	¾	1C	A ← (A) – (m…m+2)	
SUBF m	¾	5C	F ← (F) – (m…m+5)	X, F
SUBR r1, r2	2	94	r2 ← (r2) – (r1)	X
TD m	¾	E0	Test Device specified by (m)	P, C
TIX m	¾	2C	X ← (X)+1; (X) : (m..m+2)	C
TIXR r1	2	B8	X ← (X)+1; (X) : (r1)	X, C
WD m	¾	DC	Device specified by (m) ← (A) (rightmost byte)	X, F

Assembler Directives