Date: .....
Page: .....

Name: Rasad Regmi
Program: Software

Shift: Day
College Roll No: 221737

System Programming
Assignment 1

1) Explain SIC and SIC/XE.

→ SIC and SIC/XE are two hypothetical architecture that is designed to illustrate the most commonly encountered hardware concept and design.

SIC is a simpler, more straightforward architecture. Whereas, SIC/XE is an extended version of SIC, providing additional features.

- **SIC Machine Architecture**

i) Memory:
Its total memory is 2^{15} bytes.

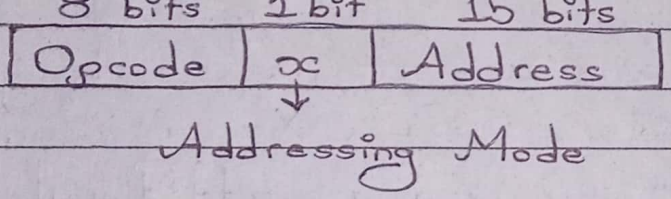
ii) Registers:
> A (Accumulator): Used for arithmetic operation
> X (Index Register): Used for indexing/addressing
> L (Linkage Register): Used to store the address return by sub-routine
> PC (Program Counter): Used to store the address of next instruction

Date :
Page :

> SW (Status word): Stores condition codes and status flags.

iii) Data Format:
IT stores integers as a 24 bit (1 word) binary number.
IT stores negative no. as a 2's complement.
IT stores character as an 8 bit ASCII code.
IT can not store floating point no.

iv) Instruction Format:



v) Addressing Mode:

x	Mode	Target Address
0	Direct Addressing	TA = Address
1	Indirect Addressing	TA = Address + (X)

vi) Instruction Set:
SIC has 7 diff. types of instruction set (i.e. Load and store, arithmetic, logical, compare, jump, subroutine, I/O).

Date: .....
Page: .....

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 right most 8 bits of register A.

- SIC/XE Machine Architecture

i) Memory:
Its total memory is 2^{20} bytes.

ii) Register:
> B (Base Register): Used for base - relative addressing
> S (General Register)
> T (General Register)
> F (Floating Point Register): Stores floating point values
Other registers are same as of SIC.

iii) Data Format:
It stores integers as a 24 bits (1 word) binary no.
It stores negative no. as a 2's complement.
It stores character as a 8 bit ASCII code.
Floating point register is used to store decimal values.

iv) Instruction Format:
~~There~~ SIC/XE uses multiple instruction formats (Format 1 to Format 4).

Date:
Page:

v) Instruction Set:
SIC/XE has 9 diff. types of instruction set (i.e. Load and store, arithmetic, floating point arithmetic operation, comparison, conditional, subroutine Linkage, register to register arithmetic, register move, I/O).

vi) Input and Output:
The I/O instructions is same as SIC.

2) Describe addressing modes used in SIC/XE.
→ SIC/XE addressing modes determine how the effective address of an operand is calculated during instruction execution.

Addressing mode ~~is~~ used in SIC/XE:
i) Immediate Addressing:
- Format: '#value'
- Indication: n=0, i=1
- Description: The operand is a constant value specified directly in the instruction.
- ~~Target Address:~~ Eg: 'LDA #5

ii) Indirect Addressing:
- Format: '@ address'
- Indication: n=1, i=0
- Description: The address field in the instruction contains the address of a memory location where effective address of operand is stored.

Date: .....
Page: .....

- Target Address: TA = M [address] (M → memory)
- E.g.: LDA @1000

iii) Direct Addressing:
- Format: 'address'
- Indication: n=1, i=1
- Description: The address field in the instruction directly specifies the memory location of the operand.
- Target Address: TA = address
- E.g.: LDA 1000

iv) Indexed Addressing:
- Format: 'address, X'
- Indication: x=1
- Description: The effective address is computed by adding the ~~value~~ contents of the index register (X) to the address field in the instruction.
- Target Address: TA = address + (X)
- E.g.: LDA 1000, X

v) Base Relative Addressing: ~~Mode:~~
- Format: 'address'
- Indication: b=1, p=0
- Description: The effective address is computed by adding the contents of the base register (B) to the displacement (address field) in the instruction.

Date:
Page:

- Target Address: TA = address + (B)
- E.g.: LDA BASE_DISP

vi) Program Counter Relative Addressing:
- Format: 'address'
- Indication: b = 0, p = 1
- Description: The effective address is computed by adding the displacement to the current value of the PC.
- Target Address: TA = address + (PC)
- E.g.: LDA PC_DISP

vii) Extended Addressing:
- Format: 'address'
- Indication: ~~e = 0, Format 3 Instruction Format~~
  e = 1, Format 4 Instruction Format
- Description: Used with Format 4 instructions to allow for 20-bit addressing. This mode allows access to a larger memory space.
- Target Address: TA = address
- E.g.: +LDA address

3) Explain RISC and CISC machines.

→ RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) are two different approaches to computer architecture design, each with its own philosophy, characteristics, and typical applications.

Date: .....
Page: .....

RISC focuses on a small set of simple instructions that execute quickly, leveraging pipelining and a Large number of registers to achieve high performance.
CISC focuses on reducing the no. of instructions per program by using complex instructions that can perform multiple tasks, potentially leading to more compact code.

- **RISC**

> Characteristics:

i) Simple Instructions
ii) Fixed Instruction Length (typically 4 bytes (32 bits))
iii) Few Addressing Modes
iv) Emphasis on Pipelining
v) Large no. of Registers

> Examples:
ARM, MIPS, SPARC, RISC-V

- **CISC**

> Characteristics:

i) Complex Instructions
ii) Variable Instruction Length
iii) Many Addressing Modes.
iv) Microprogrammed Control
v) Memory-to-Memory Operations (Can directly operate on memory)

Date: .....
Page: .....

4) Write a sequence of instruction for SIC to clear a 20 byte string to all blanks.

→

```
        START   0
        LDX     #20
        LDA     =' '
        STL     ADDR
LOOP    STCH    ADDR, X
        TIX     LOOP
ADDR    RESB    20
        END     START
ADDR    RESB    20
```

5) ~~Writ~~ With an example explain I/O operation of SIC/XE.

→ On the standard version, 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 3 I/O instructions, each of which specifies the device code as an operand.

i) Test Device (TD): tests whether device is ready or not.

ii) Read Data (RD): reads a byte from device and stores in register A.

iii) Write Data (WD): writes a byte from register A to this device.

In addition, there are I/O channels that can be used to perform input and output

Date: .....
Page: .....

while the CPU is executing other instructions.

Example: Read from an input device and write it to output device

```
INLOOP    TD      INDEV
          JEQ     INLOOP
          RD      INDEV
          STCH    DATA
          .
          .
          .
OUTLP     TD      OUTDEV
          JEQ     OUTLP
          LDCH    DATA
          WD      OUTDEV
          .
          .
          .
INDEV     BYTE    X'F1'
OUTDEV    BYTE    X'05'
DATA      RESB    1
```

This example demonstrates a typical read-write operation in SIC/XE where a byte is read from an input device and then written to an output device. The program handles device ~~readi~~ readiness using loops and performs the necessary data transfer between the CPU, memory, and I/O devices.

6) Translate (by hand) the following assembly program to SIC/XE object code. Starting address program is 1000 (H). Also assume Opcode for instruction. The output format will contains H record, T record, and E record.

→ Solⁿ; Object Code:

Location Counter	Label	Opcode	Operand	Object Code
—	STRCP2	START	1000	—
1000	FIRST	LDT	#11	741000B
1003		LDX	#0	040000
1006	MOVECH	LDCH	STR1, X	501000
1009		STCH	STR2, X	541016
100C		TIXR	T	B850
100E		JLT	MOVECH	3BFFF8
1011	STR1	BYTE	C'TEST STRING'	5445535452494E4727
1022	STR2	RESB 11		—
—		END	FIRST	—

Here we assumed opcode for each instructions. as:

LDT: 74

LDX: 04

LDCH: 50

STCH: 54

TIXR: B8

JLT: 38

Date:
Page:

Object Program:

H ^ STRCP2 ^ 001000 ^ 001017 00022
T ^ 001000 ^ 1E ^ 741011 ^ 05 ^ 000000 ^ 501011 ^
57A016 ^ 501016 ^ 3B85 ^ 38FFF8 3B2FF5
T ^ 001011 ^ 0B ^ 54454 35412053 52494E
4727
E ^ 001000

7) Explain program block with an example, a machine independent assembler feature.

→ A program block in the context of machine-independent assemblers refers to a feature that allows the assembler to handle a sequence of instructions or data as a single unit. This unit can be relocated and managed independently from other parts of the program. This feature is particularly useful for modular programming and managing large programs efficiently.

Example:

```
      USE    DATA_BLOCK
DATA1 DB     10
DATA1 RESB   10
DATA2 RESW   1234
      :      :
      USE    CODE_BLOCK
      MOV    AX, DATA1
      ADD    AX, DATA2
```

Example: Copying block of data from one ~~USE DATA_BLOCK~~ ~~DATA1~~

```
COPY        START   1000

            USE     CODE
            LDA     LENGTH
            STA     COUNT
            LDX     #0
LOOP        LDA     DATA,X
            STA     BUFFER,X
            TIXR    COUNT
            JLT     LOOP

            USE     CDATA
DATA        BYTE    C'HELLO'
LENGTH      WORD    5

            USE     BSS
BUFFER      RESB    5
COUNT       RESW    1

            USE     CODE
            END     COPY
```

Program Blocks:

The first program block named CODE contains executable instructions of program

Date: .....
Page: .....

The second named CDATA contains initialized data.
The third named BSS (Block Started by Symbol) contains uninitialized data.

Advantages:
i) Modularity
ii) Encapsulation
iii) Independence

Machine Independence:
The concept of program blocks is machine independent because it abstracts away low-level details of memory addressing and relocation. Instead of dealing directly with memory addresses, the assembler manages the placement and relocation of blocks based on higher level directives provided by the programmer. This allows the same assembly code to be assembled and executed on different computer architectures with minimal changes, as long as the assembler and linker support the abstraction.

8) Explain the following machine independent features of assembler:
- Literals
- Symbol defining statements
- Expressions

MI → Machine Independence

Date:
Page:

→• Literals:

Literals are constant values used in assembly language ~~problem~~ programs. They are defined directly in the code, typically prefixed by a specific character or set of characters. These values are often numeric constants or string constants.

In many assemblers, literals are prefixed with a specific character such as =.

Example: =5 or ='A'

> Storage: The assembler automatically ~~allocates~~ allocates storage for literals and ensures they are placed in a literal pool.

> Usage: Literals allow programmers to directly use constants without the need of for explicit definition and storage allocation.

> MI: The way literals are defined and managed is consistent across diff. machine architectures, making them a machine independent feature.

Date: .....
Page: .....

- **Symbol Defining Statements:**

It allows the programmer to define symbolic names for addresses or values. These symbols can be used throughout the assembly code to improve readability and maintainability.

Typically, symbols are defined using ~~directive~~ directives like EQU, SET, or DEFINE.

Example: MAX_LENGTH EQU 100

> Purpose: Symbols represent constants, addresses, or instruction sequences.

> Benefits: They make the code easier to understand and modify, as changes only need to be made in one place.

> MI: Symbol definitions abstract away machine specific details, making the code more portable across diff. platforms.

- **Expressions:**

Expressions in assembly language allow the use of arithmetic and logical operations to compute values. These expressions can be used in operand fields, data definitions, or other parts of the code.

Expressions combine symbols, literals, operators, and possibly function calls.

Example: LENGTH = END - START

Date:
Page:

> Evaluation: The assembler evaluates expressions during the assembly process to resolve values before the machine code is generated.

> Flexibility: Expressions provide flexibility in defining values that depend on other constants or labels, allowing for more dynamic and adaptable code.

> MI: The rules for writing and evaluating expressions are consistent across diff. assemblers, contributing to their machine - independent nature.

9) Explain control section and program linking.

→ Control Section:

A control ~~sef~~ section, often abbreviated as CSECT, is a segment of code or data that is treated as a single unit by the assembler and linker. Each control section can be assembled independently and can be located anywhere in the memory when the program is loaded.

> Modularity:

Control sections enable the division of a

Date :
Page :

program into independent modules.

~~> Isolat~~

> Isolation:
Each control section has its own symbol table, which means symbols defined in one section are not visible in another unless explicitly shared.

> Linking:
The linker combines multiple control sections into a single executable program, resolving references bet^n them.

> Relocation:
Since control sections are relocatable, they can be placed at diff. memory locations without altering the code.

> Program Linking:
Program Linking is the process of combining multiple object files (produced by assembling control sections) into a single executable program. The linker resolves references bet^n diff. control sections and ensures that all external references are correctly addressed.

> Object files:
Assemblers produce object files containing

machine code and relocation information for each control section.

> Linker:
The linker takes these object files and combines them, resolving symbol references and addresses.

> External References:
Symbols defined in one control section but used in another are resolved by the linker through external references

> Relocation:
The Linker adjusts the addresses of symbols and instructions based on the final memory layout of the program.

> Library Linking:
In addition to user-defined object files, the linker can include code from standard or user-defined libraries

> Static vs Dynamic Linking:
→ Static Linking: All required modules and libraries are combined into a single executable at compile time.

Date: .....
Page: .....

→ Dynamic Linking: Some modules or libraries are linked at runtime, allowing for smaller executables and shared library usage.

10) Write LOCCTR, Object Code and Object program for the following:

→ Object Code:

Location Counter	Symbol	Opcode	Operand	Object Code
1000	$	START	1000	—
1000	FIRST	STL	RETADR	141027
1003		LDX	#LENGTH	041033
1006		JSUB	RDREC	482039
1009		JSUB	CONVRT	482057
100C		STA	NUM1	0C102A
100F		JSUB	RDREC	482039
1012		JSUB	CONVRT	482057
1015		STA	NUM2	0C102D
1018		LDA	NUM1	00102A
101B		ADD	NUM2	181020
101E		STA	RESULT	0C1030
1021		JSUB	WRREC	482073
1024		J	@RETADR	3C1027
1027	RETADR	RESW	1	—
102A	NUM1	RESW	1	—
102D	NUM2	RESW	1	—
1030	RESULT	RESW	1	—
1033	LENGTH	WORD	6	000006

Date : .....

Page: .....

Location Counter	Symbol	Opcode	Operand	Object Code
1036	BUFFER	RESB	6	—
—		END	FIRST	—
103C	RDREC	CLEAR	X	B410
103E		CLEAR	A	B400
1040		CLEAR	S	B440
1042	RLOOP	TD	INPUT	E32080
1045		JEQ	RLOOP	332042
1048	RD	INPUT	X	C2000
104B		COMP	X,#6	A30006
104E		JEQ	EXIT	332054
1051		STCH	BUFFER,X	542036
1054		TIX	LENGTH	2C1033
1057		JLT	RLOOP	383042
105A	EXIT	STX	LENGTH	101033
105D		RSUB		4F0000
1060	WRREC	CLEAR	X	B410
1062		CLEAR	A	B400
1064		CLEAR	S	B440
1066	WLOOP	TD	OUTPUT	E32005
1069		JEQ	WLOOP	332066
106C		LDCH	BUFFER,X	102036
106F		WD	OUTPUT	DC2005
1072		TIX	LENGTH	2C1033
1075		JLT	WLOOP	382066
1078		RSUB		4F0000

Date : .....

Page : .....

Location Counter	Symbol	Opcode	Operand	Object Code
107B	CONVRT	CLEAR	A	B400
107D		CLEAR	X	B410
107F		LDX	#0	041000
1082	CLOOP	TD	BUFFER,X	E32036
1085		JEQ	CEXIT	332088
1088		ADDR	A,X	180010
108B		LDA	BUFFER,X	002036
108E		SUB	#C_ZERO	1C3030
1091		STA	BUFFER,X	0C2036
1094		TIX	LENGTH	2C1033
1097		JLT	CLOOP	382082
109A		CXIT	RSUB	4F0000
109D	INPUT	BYTE	X'F1'	F1
109E	OUTPUT	BYTE	X'05'	05
109F	C_ZERO	BYTE	X'30'	30

- SYMTAB:

SYMBOL	LOCCTR
FIRST	1000
RETADR	1027
NUM1	102A
NUM2	102D
RESULT	1030
LENGTH	1033
BUFFER	1036
RDREC	103C

Date : .....
Page: .....

SYMBOL	LOCCTR
RLOOP	1042
RD	1045
EXIT	1057
WREREC	1060
WLOOP	1066
CONVRT	107B
CLOOP	1082
CXIT	109A
INPUT	109D
OUTPUT	109E
C-ZERO	109F

- Object Program:

H^FIRST_^001000^001083 00009F
T^001000^1E^141027^041033^
482000^ 48203C^482057^0C102A
^48203C^482057^0C1020^00102A^
181020^0C1030^482073^3C1027
T^001033^26^000006^B410^B400
^B440^E32080^332042^C2000^
A30006^332054^542036^2C1033^
383042^101033^4F0000
T^001060^19^B410^B400^B440
^E32005^332066^102036^
DC2005^2C1033^382066^4F0000
T^00107B^1F^B400^B410^041000
^E32

Date : .....
Page: .....

^E32036^332088^180010^002036^
1C3030^0C2036^2C1033^382082^
4F0000
T^001089^06^F1^05^30
E^001000

11) Relocate the above program into diff. memory of your choice and write the resulting object program with the suitable modification record.

→ Sol-n; Let new starting address be 2000,

Original starting address: 1000
New starting address: 2000
Displacement: 2000 - 1000 = 1000

- Object Program:

H^FIRST_^002000^00009A^F
T^002000^1E^141027^041033^48203C
^482057^0C202A^48203C^482057^
0C202D^00202A^18202D^0C2030^
482073^302027
T^002033^26^000006^B410^B400^
B440^E32080^332042^C20000^A30000E
^332054^542036^2C2033^383042^
102033^4F0000
T^002060^19^B410^B400^B440^
E32005^332066^102036^DC2005^
2C2033^382066^4F0000

Date : .....

Page: .....

T^002078^25^B400^B410^041000
^E32036^332088^18000^002036
^1C3030^0C2036^2C2033^382082
^4F0000^F1^05^30
M^002001^06^+FIRST
M^002004^06^+FIRST
M^002007^06^+FIRST
M^00200A^06^+FIRST
M^00200D^06^+FIRST
M^002010^06^+FIRST
M^002013^06^+FIRST
M^002016^06^+FIRST
M^002019^06^+FIRST
M^00201C^06^+FIRST
M^00201F^06^+FIRST
M^002022^06^+FIRST
M^002025^06^+FIRST
M^002042^06^+FIRST
M^002045^06^+FIRST
M^002048^06^+FIRST
M^00204B^06^+FIRST
M^00204E^06^+FIRST
M^002051^06^+FIRST
M^002054^06^+FIRST
M^002057^06^+FIRST
M^00205A^06^+FIRST
M^002066^06^+FIRST
M^002069^06^+FIRST

M^002060^06^+FIRST
M^002066^06^+FIRST
M^002072^06^+FIRST
M^002075^06^+FIRST
M^00207F^06^+FIRST
M^002082^06^+FIRST
M^002085^06^+FIRST
M^00208B^06^+FIRST
M^00208E^06^+FIRST
M^002091^06^+FIRST
M^002094^06^+FIRST
M^002097^06^+FIRST

~~13)~~ ~~12)~~ 14) Define CSECT, EXTDEF and EXTREF.

→. CSECT:

A CSECT or Control Section, is a block of code or data that can be seperately assembled, compiled, and linked. Each control section represents an independent unit of an object program.

It allows modular programming and seperate compilation, making it easier to manage large programs. Each CSECT can be independently developed and maintained.

CSECT is used to define segments of a program that can be loaded and relocated independently.

Page: .....

- **EXTDEF:**

EXTDEF or External Definition, is a declaration that makes a symbol ~~fvariabt~~ defined in one control section available to other control sections.

It is used to share symbols across differe control sections, enabling modularity and ~~e~~ interdepency bet- different parts of a program.

When a symbol is defined in one CSECT and needs to be referenced in another, EXTDEF is used to declare the symbol for external linkage.

- **EXTREF:**

EXTREF or External Reference, is a declaration that indicates a symbol used within a control section but defined in another control section.

It allows a control section to refer to symbols that are defined elsewhere, facilitating the linking of separately ~~ass~~ assembled or compiled modules.

EXTREF is used within a CSECT to

Date: .....
Page: .....

reference symbols that are externally defined, ensuring proper linking during the assembly or linking process.

15) Explain Load and go assembler.

→ A Load and go assembler is a type of assembler that translates assembly Language code into machine code and immediately loads it into memory for execution, without producing an intermediate object file or requiring a seperate linking step.

Key Features:

i) Immediate Execution:
After assembling the source code, the load and go assembler directly loads the generated machine code into memory and transfers control to it for execution.

ii) No Intermediate Object Files:
Unlike traditional assemblers that produce object files to be linked later, a Load and go assembler skips this step, directly producing executable code in memory.

iii) Simplicity:
The process is simplified because it
~~etc~~

eliminates the need for seperate Linking and Loading phases.

iv) Integrated Environment:
Load and go assemblers are often part of an integrated development environment (IDE) or a development system where the assembly, loading, and execution are handled within a single tool.

Advantages:
i) Speed
ii) Convenience for Development
iii) Reduced Complexity

Disadvantages:
i) Limited Modularity
ii) Memory Usage
iii) Lack of Flexibility

Example:
An embedded systems development environment where a dereloper writes assembly code for a microcontroller, and the development toolchain assembles the code and immediately loads it into to microcontroller for testing.

Date:
Page:

10) Explain multipass assembler.

→ A multipass assembler is an assembler that processes the source code multiple times (passes) to resolve all symbols, addresses, and dependencies before generating the final machine code. The need for multiple passes arises from the complexity of handling forward references, where instructions might reference labels or variables not yet defined in the source code.

- How Multipass Assemblers Work?

> Pass 1:

i) Symbol Table Creation:
The assembler builds a symbol table, which is essentially a dictionary of symbols and their locations.

ii) Address Calculation:
The assembler calculates the addresses of instructions and data locations, but does not generate the final machine code yet.

iii) Syntax Checking:
Basic syntax checking is performed during this pass to ensure that the code follows the assembly language's syntax rules.

Page: .....

> Pass 2 (and subsequent passes if needed):

i) Code Generation:
With the symbol table available, the assembler can now replace symbols with their corresponding addresses and generate the actual machine code.

ii) Resolution of Forward Reference:
Any forward references identified during the first pass are now resolved using the symbol tabl

iii) Final Assembly:
The final machine code is produced, including a necessary relocations and linkages.

- Advantages:

i) Forward Reference Handling
ii) Optimized Code
iii) Flexibility

- Disadvantages:

i) Time Consuming
ii) Resource Intensive