Internal Assessment Test III – May 2018

Scheme and Solution

Sub: System Software & Compiler Design (15CS63)

Q.1. Explain the SIC/XE machine architecture.
Ans:

SIC/XE machine architecture

- **Memory**
- **Registers**
- **Data Formats**
- **Instruction Formata**
- **Addressing Modes**
- **Instruction Set**
- **Input and Output**

CMR

- SIC/XE machine architecture

- **Memory :-**

Memory structure for SIC/XE is same as SIC. The maximum memory on SIC/XE is 1 mega byte (MB) → 2^{20} bytes

↳ this increase in memory leads to changes in instruction formats and addressing modes.

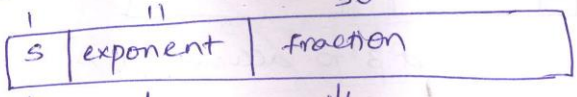
- **Registers :-**

Apart from 5 SIC registers the following additional reg are provided by SIC/XE.

mnemonic	Number	use
B	3	Base reg. used for addressing
S	4	General working reg
T	5	General working reg
F	6	floating pt accumulator

- **Data formats :-**

SIC/XE provides the same data formats as in SIC. In addition there is a 48 bit floating point data type with the following format.



↓ sign — 0 for positive, 1 – negative

↓ unsigned binary no b/n 0 and 2047.

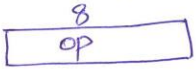
↓ value b/n 0 and 1

→ If the exponent has value e and fraction has value f then the absolute value of the number is represented as

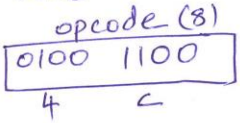
$f * 2^{(e-1024)}$ [$\pm$ mantissa * 2^{exp}]

- Instruction formats

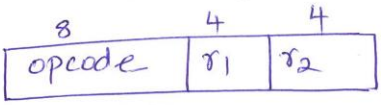
- Format 1 (1byte)



Ex :- RSUB ——— opcode is 4C



- Format 2 (2 bytes)

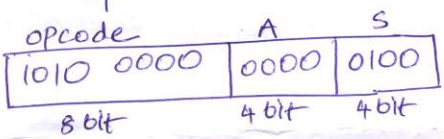


Ex :- comp A, S (compares the contents of reg A and S)

Hex to binary

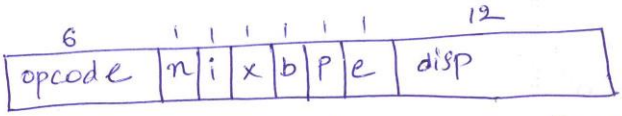
$(A)_{16} = (1010)_2$

Lopcode · A 0

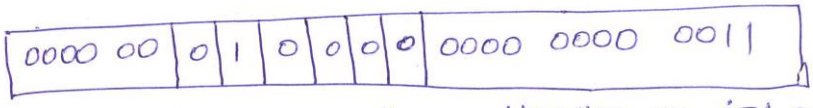


A 0 0 4 —— object code

- Format 3 (3 bytes)

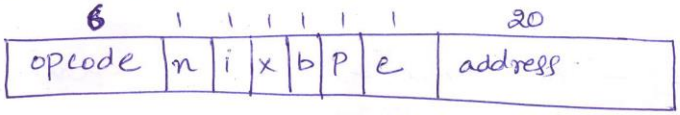


Ex :- LDA #3 (Load 3 to accumulator A)



- Here # represents it is immediate addressing so i=1, n=0 and the target address itself is used as operand value and no memory ref is performed.

- Format 4 (4 bytes)



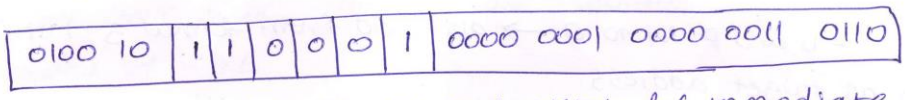
Flags :- n=0, i=1 → immediate addressing – TA is used as an operand value

– n=1, i=0 → indirect addressing

Ex :- + JSUB RDREC (JUMP to the address 1036)

JSUB - 48

2|8
2|4 - 0
2|2 - 0
2|1 - 0



→ By default n and i are set to 1 except for immediate and indirect modes.

→ b and p are set to 1 and 0 only in base and pc modes

- Addressing modes

– Base relative addressing → disp field is format 3 instsn with 12 bit.

L b=1, P=0 ; TA = (B) + disp

~~– PC relative b=0, P=1 → TA = (PC) + disp~~

Ex :- 1056 STX LENGTH

↳ opcode 10

→ Direct addressing mode (a, b, p → set to 0)

↳ n, i = 1, 0 for format 3

→ Relative :- addr of the operand should be added to the current value stored at B reg. (b=1) & to the value stored at PC reg. (PC=1) – Base relative, b=1, P=0 TA = (B) + disp

– PC relative, b=0, P=1 TA = (PC) + disp

Q.2. Generate object code for the given SIC/XE program 10M

```
QUIZ    START   0
FIRST   LDA     #3
        STX     THREE
        LDX     #0
        +LDS    THREE
        ADDR    A,X
        +STA    RESULT,X
RESULT  RESW    1
THREE   RESW    1
        END     FIRST
```

OPCODES:

LDA-00

STX-10

LDX-04

LDS-6C

ADDR-90

STA-0C

Object Code

Location	Label	Mnemonic	Operand	Object Code
	QUIZ	START	0	
0000	FIRST	LDA	#3	010003
0003		STX	THREE	132010
0006		LDX	#0	050000
0009		+LDS	THREE	6F300009
000D		ADDR	A,X	90
000F		+STA	RESULT ,X	0FB00013
0013	RESULT	RESW	1	
0016	THREE	RESW	1	
0019		END	FIRST	

Q.3 Define program relocation. Explain the different ways of doing program relocation.

Ans: Relocation is the process of assigning load addresses to position-dependent, but locatable code of a program and adjusting the code and data in the program to reflect the assigned addresses.

One problem faced by users of simple assembly and loading systems is where to put programs in memory. Up to this point in the discussion, it has been assumed that assembly programs would start at location zero unless the programmer explicitly set a different assembly origin by

modifying the location counter. This is clearly not an acceptable solution on large systems where code must be written without any knowledge of where it will eventually be run. The solution is to introduce a *relocation mechanism*, that is, a mechanism that allows the decision about where to put a program in memory to be deferred until after the program has been assembled and compiled.

If we require the program to be reassembled in order to change the location where it will be loaded, we could refer to this as *assembly-time relocation*, and this view is useful, particularly if the source code was compiled and the assembly code itself is viewed as something like an object code for communication between the compiler and a load and go assembler.

At the other extreme, some machine languages allow a running program to be unloaded from memory and reloaded in a different location without harm. This is called *run-time relocation*. Run-time relocation is only possible if consistent use is made of base registers or relative addressing. If the machine code or the user data structures contain absolute memory addresses, moving the code and data to a different address would be very difficult, but if all branch addresses and pointers are relative, that is, expressed as displacements from the memory location containing the address to the memory location to which the address refers, we can move the entire block of data holding a program's code and data to a different memory address. Machines that allow programs to be written this way are said to support *position independent code*.

Virtual memory hardware (to be discussed later) can hide the complexity of position indepent programming from users by using special address translation hardware to perform run-time relocation, but usually, ther term position independent code is reserved for programs which are explicitly coded to be able to be run at any memory address without the need to edit any memory addresses in the code.

Another way of thinking of relocation is in terms of when the objects that make up a program are bound to actual memory locations. We can therefore speak about the *binding time* of each object in the program. Some objects may be explicitly bound by the programmer. Some objects may be bound to specific memory locations at compile time, *compile-time binding*, and the binding of some objects may be deferred, for example, until the time the program is loaded in memory or even later, in *run-time binding*.

If binding is done at any time between run-time, when it is appropriate to speak of position independent code, and assembly time or compile time, we must modify the object code to allow a distinction between the values that are already determined -- those representing constants and those representing addresses of objects that are already bound to specific memory addresses, and values that are not-yet bound.

Q.4. With an algorithm, explain pass-1 of a linking loader. 10M

Ans:

Algorithm for Pass 1 of a linking loader:

```
Pass 1:
Begin
get PROOADDR from operating system
set CSADDR to PROOADDR {for first control section}
while not end of input do
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type ~ 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                    begin
                        search ESTAB for symbol name
                        if found then
                            set error flag (duplicate external symbol)
                        else
                            enter symbol into ESTAB with value(CSADDR +
                        indicated address)
                        end {for}
                end {while ~ 'E'}
            add CSLTH to CSADDR {starting address for next control section}
        end {while not EOF}
    end {Pass 1}
```

Q.5. Write and explain the algorithm for a pass-1 of two-pass assembler. 10M

Ans: <u>Algorithm for Pass 1 assembler</u>:

```
begin
        if starting address is given
                LOCCTR = starting address;
        else
                LOCCTR = 0;
        while OPCODE != END do              ;; or EOF
                begin
```

```
        read a line from the code
        if there is a label
            if this label is in SYMTAB, then error
            else insert (label, LOCCTR) into SYMTAB
        search OPTAB for the op code
        if found
            LOCCTR += N          ;; N is the length of this instruction (4 for MIPS)
        else if this is an assembly directive
            update LOCCTR as directed
        else error
        write line to intermediate file
        end
    program size =  LOCCTR - starting address;
end
```

Q.6. (a) Give the target address generated for the following machine instruction: **6M**

i) 032600h (ii) 03C300h (iii) 0310C303hif (B)=006000, (PC)=003000, (X)=00090

Ans: (i) Target Address= 600+3000=3600H

(ii) TA is 300+90+6000= 6390H

(iii) TA is 0C303

(b) Write the differences between system software and application software with examples.

4M

Ans:

System software	Application software
1. machine dependent	1. machine independent.
2. Focus is on the computing system relating to architecture on which they run.	2. Focus is on application.
3. support the operation and use of computer itself rather than any particular application	3. primarily concerned with solution of some problem using computer as a tool.
Ex:- Assemblers, compilers, operating system	Ex:- Adobe, ms word etc.

7. What is loader? What are its advantages and disadvantages? Explain the boot strap loader with algorithm. 10M

Ans:

In computer **systems** a **loader** is the part of an operating **system** that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a **program**, as it places programs into memory and prepares them for execution.

Alternatively referred to as bootstrapping, bootloader, or **boot** program, a **bootstrap loader** is a program that resides in the computer's EPROM, ROM, or other non-volatile memory. It is automatically executed by the processor when turning on the computer.