KINZA AFZAL 23k-0842

COAL ASSIGNMENT

Q1) Differences / Relation

ASSEMBLY LANGUAGE	HIGH LEVEL LANGUAGE
1- It needs an assembler for conversion	It needs a compiler/interpreter for conversion
2- It is machine dependent.	It is machine independent.
3- In this, mnemonics, codes are used.	In this, statements similar to English language are used.
4- It supports low level operation.	It doesnot support low level languag
5- More compact code.	No compactness.
6- In this, it is easy to access hardware component.	In this, it is difficult to access hardware component.
7- Assembly language codes require less memory space.	High-level language codes requir more memory space.
8- Code is relatively difficult to debug.	Code is relatively easy to debug.
9- The readability of assembly language codes is less.	High-level language codes are more readable.
10- Less user friendly.	More user friendly.
11- Assembly language programs take more time and effort to develop.	High level language programs require less development time and effort.
12- It is primarily used to program processors, microcontrollers, embedded systems, device drivers e.t.c	It is mainly used for developin software applications, web applications and more e.t.c.

SIMILARITIES :-

① Both ultimately break down to machine code .i.e 1's and 0's.

② Both share the same algorithm for a given task, as algorithms are language independent.

③ Both are used by programmers.

④ Both have mechanisms for error detection and handling.

⑤ Both can perform operations on data, such as arithmetic operations, data storage and retrieval.

⑥ Both have defined syntax and semantics that dictates how programs are written and understood by computer.

⑦ Both incorporate control structures such as loops, conditionals and functions to manage flow of execution.

b) Assemblers and Compilers.

COMPILER	ASSEMBLER
① Translates high level language into machine code.	Translates assembly language into machine code.
② Translates all code at the same time.	Uses the processors instruction set to convert.
③ Output of compiler is a mnemonic version of machine code	Output of assembler is binary code.
④ Compiler input is preprocessed source code.	Assembler inputs source code.
⑤ Compiler is more intelligent than assembler.	Assembler is less intelligent than compiler.

COMPILER	ASSEMBLER
(6) Memory requirement is more due to creation of object code. ~~GAS and GNU~~	It ~~requires~~ generate object code by converting symbolic op-code into numeric op-code.
(7) C, C++, C# and Java uses compiler.	GAS and GNU uses assembler.

(c) Why HLL are more portable than assembly language?

Assembly language is not portable because it is designed for a specific processor family. Assembly language is machine dependent whereas high level language is machine independent. This is because high level languages abstract away hardware details, making them portable across different systems while assembly languages are specific to particular hardware architectures and thus not portable. [as mentioned in (a) & (b)]. Assembly language depends on machine code instructions, i.e :- every assembler has its own assembly language.

Examples:-

(1) Game programmers use assembly language to take full advantage of specific hardware features in a target system.
Ex :- Some games are available only on iOS devices.

(2) Embedded system programs are written in assembly language & downloaded into computer chips and

installed in dedicated devices.

Ex:- air conditioning control systems, modems, ignition systems e.t.c.

③ A program written in C (a HLL) can be compiled and run on different operating systems and hardware platforms.

Ex:- same C code can run on windows, Linux or macOs and on different CPU architectures like x86, or ARM.

Q.2.)

Level3:- It is the high level language program written in C. It is a user friendly language, and easier to understand.

Level2:- Level 2 is assembly language, which is a low-level language that is close to machine language. It uses mnemonics like call, sbi, rjmp e.t.c. It is used to write programs that are efficient.

Level1:- Level 1 is machine level, which is level of machine code. The computer can execute machine code directly. It is the Instruction Set Architecture.

Level0:- Level0 is the hardware level. This includes the CPU, memory and input/output devices.

Level 0 has Arduino UNO board, pins, resistors e.t.c which are the hardware components.

VM [Virtual Machine]:- is computer resource that uses software instead of physical computer to run programs and deploy apps.

Q2) VM0:- VM0 is responsible for translating assembly code into machine code and executing it on the Arduino's processor (level 1) to execute on physical hardware (level 0).

VM1:- The C code (level 3) is compiled using compiler. This converts high level C code into low level Assembly language (level 2).

→ Compilation occurs at VM1 (Assembly language level).

→ Translation to machine code occurs at VM0 (hardware level).

Q3) Roll no :- 23K-0842

Last 4 digits = 0842

MOV EAX, F0F8F4F2h
ADD EAX, 10000100h

Both are hexa

F = 15

Addition :-

a)

```
  F  0  F  8  F  4  F  2
+ 1  0  0  0  0  1  0  0
-------------------------
1  0  0  F  8  F  5  F  2
```

Ans: 100F8F5F2h

Carry Flag (CF)	1
Overflow Flag (OF)	0
Zero Flag (ZF)	0
Sign Flag (SF)	0

Q3)

Parity Flag (PF)	0
Auxilary Carry Flag (AF)	0

b. 8086 uses a segmented memory model. These registers (CS, DS, SS and ES) can hold 16-bit values. These segment registers define the starting address of 64KB segment in memory.

Segment Registers

CS
DS
SS
ES

16-bit Segment Registers

offset value (also 16-bit is added) to segment address to form physical memory address.

Physical Address = (Segment register value * 16) + Offset

Maximum Segment register value :- 2^{16} = 65,535 bits (FFFFh)

Maximum offset = 2^{16} = 65,535 = (FFFFh)

= (65,535 * 16) + 65,535 = 1114095 or 10FFEFh

However the physical address space is limited to 2^{20} bytes. This is because the segment base address can shift the segment by upto 16 bytes, but combining all segment addresses & offsets the total addressable space doesnot exceed 1MB.

1MB = 1024 * 1024 = 1048576 bytes

$2^{16} * 2^4 = 2^{20}$ = 1048576 bytes

Q3) c. 12AB : 025F

Seg : offset

Physical Address = (seg x 10h) + offset

= (12AB x 10h) + 025F

= 12AB0 + 025F

= 12D0Fh

```
  12AB0
+ 0025F
-------
  12D0F
```

d. The segment registers in 8086 are 16 bits each.

Address refers to a single location in memory, and x86 processors permit each byte location to have a separate address. This is called byte-addressable memory. The linear or (absolute) address is 20 bits ranging from 0 to FFFFF hexadecimal. Programs cannot use linear addresses directly so address are expressed using two 16-bit integers.

Q4) INCLUDE Irvine32.inc

```
.data
SUNDAY    EQU 0
MONDAY    EQU 1
TUESDAY   EQU 2
WEDNESDAY EQU 3
THURSDAY  EQU 4
FRIDAY    EQU 5
```

```
SATURDAY  EQU 6
weekdays  BYTE  "SUNDAY", "MONDAY", "TUESDAY", "WEDNESDAY",
          "THURSDAY", "FRIDAY", "SATURDAY",

.code
main PROC
call Dumpregs
 exit
main ENDP
END main
```

Q5) INCLUDE Irvine32.inc

```
.data
    V1  DWORD  5 DUP(?)
    V2  BYTE   2 DUP(?)
    V3  BYTE   15 DUP('&')
    V4  BYTE   7 DUP('%')
    V5  BYTE   1 DUP('M')

.code
main PROC
call Dumpregs
 exit
main ENDP
END main
```

Q6) mov al, 88h
add al, 90h

```
 88
+90
---
118
---
carry
```

8+9 = 17
16-17 = 11

al = 18h
CF = 1
OF = ~~01~~ 1

CF is 1 because there is a carry generated.
OF is 1 as the result exceeds the size of register.

mov al, 5
add al, 123
5 + 123 = 128 in decimal

Decimal to hexadecimal :-
128 → Binary

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0

1000 0000 → hexa
(8421) (8421)
8 0 = 80h

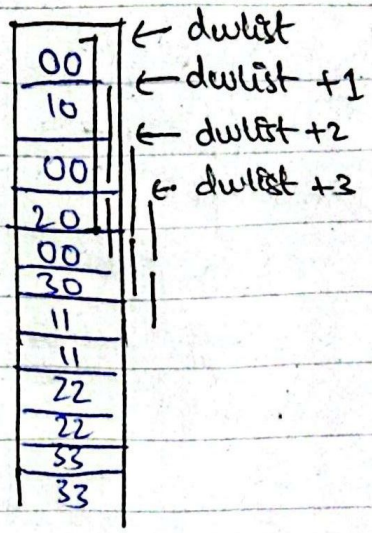
al = 80h
SF = 1, OF = ~~0~~ 1, CF = 0
CF is 0 because no carry generated.
SF is 1 because MSB bit is 1.
OF is 1 because for an 8 bit register, the range is -128 to 127. Since 128 is outside this range and 80h (hexa decimal) represents a negative number in 2's complement notation.

Q7)

```
mov eax , dwList          eax = 20001000
mov ebx , [dwlist +1]     ebx = 00200010
mov ecx , [dwlist +2]     ecx = 30002000
mov edx , [dwlist +3]     edx = 11300020
```

dwlist → DWORD
↓
4 bytes

memory map



mov eax, dwlist, here it is at the starting address hence value in bytes is 20 00 10 00
↓
total 4 bytes

dwlist + 1 doing this it moves 1 byte ahead

00200010

dwlist + 2

30002000

dwlist +3

11300020

* Justified by memory map.