**MP Oral**

1. What is use of defining Macro? A **macro** (which stands for "macroinstruction")

An assembly language macro is a predefined set of instructions that can easily be inserted wherever needed. Once defined, the macro can be used as many times as necessary. It is useful when the same set of code must be utilized numerous times. A macro can be useful to reduce the amount of coding, streamline programs, and reduce errors from repetitive coding.

In the C Programming Language, the #**define** directive allows the **definition** of **macros** within your source code. These **macro definitions** allow constant values to be declared for **use** throughout your code. **Macro definitions** are not variables and cannot be changed by your program code like variables.

**2:** What is difference between instruction & Directive:

An **instruction** is directly translated to something the CPU can execute. A **directive** is something the assembler can interpret, and tells something about the way the **instructions** are to be assembled.

3: What is structure of assembly language program?

An **assembly language program** is a series of statements, which are either **assembly language instructions** such as ADD and MOV, or statements called directives. A square bracket ( [ ] ) indicates that the field is optional. The label field allows the **program** to refer to a line of **code** by name.

4: Difference between . data and . bss section

**BSS** refers to uninitialized global and static objects and **Data** refers to initialized global and static objects. Both **BSS** and **Data** usually refer to RAM objects.

5: What is function of value 10d, 13d while declaring message?

These character are used to control the cursor position. The 10 shifts cursor on new line with same column no. and 13 returns the cursor to its initial position of line that is at start of the line! So, every time you make use of these control characters as the part of string or any array.

6: Differentiate between DB and RESB directives

**DB** allocates in chunks **of** 1 byte. DW allocates in chunks **of** 2 bytes. ... **RESB** 1 allocates 1 byte. RESW 1 allocates 2 bytes.

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. The current pseudo-instructions are DB, DW, DD, DQ and DT, their uninitialised counterparts RESB, RESW, RESD, RESQ and REST, the INCBIN command, the EQU command, and the TIMES prefix.

**7: Explain JS and JNS Instructions.:**

This is performed by a set of jump **instructions** j<condition> depending upon the condition

|  |  |  |
| --- | --- | --- |
| **JS** | Jump Sign (negative value) | SF |
| **JNS** | Jump No Sign (positive value) | SF |

What is JNS instruction?

Jump if not signed (Jump if positive) Description: Jumps to the destination label mentioned in the **instruction** if the SF is set, else no action is taken. If the sign flag is 0 it indicates a positive signed number. Hence the **instruction** causes a jump if the result of previous **instruction** is positive.

* he js (or jne) instruction is a conditional jump that follows a test.
* It jumps to the specified location if the previous instruction sets the [Sign Flag (SF)](https://www.aldeid.com/wiki/X86-assembly/Registers#SF_.28Sign_Flag.29).

Syntax

js *location*

**8: Which registers are used to point to source and destination arrays?**

**May be general purpose registers.**

**9:** **What is exit function in 64 bit syscall?**

On many computers operating systems, a computer process terminates its execution by making

The **exit** operation typically performs clean-up operations within the process space before returning control back to the operating **system**.

**10: How push and pop instructions are executed?**

SP points to current stack top. By default, the stack grows downward in memory, so newer values are placed at lower memory addresses. To **push** a value to the stack, the **PUSH instruction** is used. To **pop** a value from the stack, the **POP instruction** is used.

**11: What are different addressing modes to specify an address in an instruction?**

The basic addressing modes are: • Register Mode Addressing • Immediate Mode Addressing

• Memory Mode Addressing

Or:

* 2) Index **Mode**. The **address** of the operand is obtained by adding to the contents of the general register (called index register) a constant value. ...
* 3) Indirect **Mode**. ...
* 4) Absolute (Direct) **Mode**. ...
* 5) Register **Mode**. ...
* 6) Displacement **Mode**. ...
* 7) Autoincrement /Autodecrement **Mode**.

**12:** **Explain an instructions: movsb, std, cmps, cld, jbe ,rol, loop:**

**X86 has 1503 instructions.**

In specific, **movsb** copies one byte from ds:esi to es:edi , the adjusts both esi and edi by 1, either up or down depending on the direction flag.

# CLD — Clear Direction Flag: Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

# Cmps : compare string operands

# Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

# Loop : loop according to ECX counter

# Performs a loop operation using the RCX, ECX or CX register as a counter

# Movsb : Move data from string to string

# Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand).

# Rol : rotate;

# Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand.

# Std: set direction flag

# Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

**13:** **What is logic to convert Hex to Bcd and vice versa**

**To convert** (213AFE)H to **BCD**, first it has to be converted to binary which gives (2177790)D . Now each digit is converted to its **BCD code** which gives (0010 0001 0111 0111 0111 1001 0000)**BCD**

**14: What is the near and far reference of procedure?**

**NEAR**: Defines a **near procedure**; called with LCALL or ACALL. **FAR**: Defines a **far procedure**; called with ECALL. You should specify **FAR** if the **procedure** is called from a different 64KByte segment.

A **near** call refers a **procedure** which is in the same code segment. A **Far** call refers a **procedure** which is in different code segment It is also called Intra-segment call. It is also called Inter-segment call A **Near** Call replaces the old IP with new IP A **FAR** replaces CS & IP with new CS & IP.

**15: Which bit is responsible for switching from real to protected mode?**

32-**bit** segment offsets. Ability **to switch** back to **real mode** without resetting. Virtual 8086 **mode**.

**16: List names of Protected mode specific registers along with their size:**

**The** six segment **registers** available in **80386** are CS, SS, DS, ES, FS and GS. **The** CS and SS are **the** code and **the** stack segment **registers** respectively, while DS, ES, FS, GS are 4 data segment **registers**. A 16 bit instruction pointer IP is available **along** with 32 bit counterpart EIP.

**17: What is the difference between load & store operations of registers?**

**Load** instructions move data from memory to **registers**. **Store** instructions move data from **registers** to memory.

**18: What is Recursion? How to use stack in recursive functions?**

**Recursive functions use** something called “the call **stack**.” When a program calls a **function**, that **function** goes on top of the call **stack**. This similar to a **stack** of books. You add things one at a time. Then, when you are ready to **take** something off, you always **take** off the top item.

**19: What are features of 80387 co processor?**

Intel **80387** is a **co-processor** for 80386 microprocessor family.  
...  
**Being fully object-code compatible with 80287 and 8087 co-processors, the 80387 has much greater performance due to a few factors:**

* Higher **co-processor** clock frequency.
* Execution of optimized FPU instructions requires fewer clock cycles.
* Data bus is 32-bit.

**20:** **What is a Microprocessor?**

**Microprocessor**, any of a type of miniature electronic device that contains the arithmetic, logic, and control circuitry necessary to perform the functions of a digital computer's central processing unit.

**21: What is Instruction Set?**

An **instruction set** is a group of commands for a **CPU** in machine language. ... All CPUs have **instruction sets** that enable commands to the **processor** directing the **CPU** to switch the relevant transistors. Some **instructions** are simple read, write and move commands that direct data to different hardware.

Three **types of instruction** are: 1-byte **instruction**, 2-byte **instruction**, and 3-byte **instruction**. In 1-byte **instruction**, the opcode and the operand of an **instruction** are represented in one byte.

**22: What are the features of Intel 80386**

**80386** has data bus of 32-bit. It holds address bus of 32 bit. It supports physical memory addressability of 4 GB and virtual memory addressability of 64 TB. **80386** supports variety of operating clock frequency, which are 16 MHz, 20 MHz, 25 MHz and 33 MHz.

**23: What is Logical Address?**

**Logical Address** is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program’s perspective.  
The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

**24: What is Physical Address?**

**Physical Address** identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.

**25: What are the flags in 80386?**

|  |  |  |  |
| --- | --- | --- | --- |
| OF | nv/ov | Overflow |  |
| DF | up/dn | Direction | Must be *up* at function boundaries |
| SF | pl/ng | Sign |  |
| IF | ei/di | Interrupts | Set if interrupts are enabled |
| ZF | nz/zr | Zero |  |
| AF | na/ac | Auxiliary carry | Not used by C code |
| PF | pe/po | Parity | Not used by C code |
| CF | nc/cy | Carry |  |