CT-1 marks-30

1. Discuss different categories of functions performed by a microprocessor.

Answer:
Microprocessors are versatile electronic devices that can perform a wide range of functions. Here are some of the different categories of functions typically performed by a microprocessor:

1. Arithmetic and Logic Operations: Microprocessors excel at performing arithmetic calculations (such as addition, subtraction, multiplication, and division) and logical operations (such as AND, OR, NOT, and XOR). These operations are fundamental to many computational tasks.

2. Control Unit Operations: Microprocessors have a control unit that coordinates and manages the execution of instructions. This involves fetching instructions from memory, decoding them, and generating the necessary control signals to execute them. The control unit ensures the proper sequencing and synchronization of operations within the microprocessor.

3. Memory Operations: Microprocessors interact with different types of memory, including RAM (Random Access Memory) and ROM (Read-Only Memory). They can read data from memory and write data to memory locations. Memory operations are essential for storing and retrieving data and instructions during program execution.

4. Input/Output (I/O) Operations: Microprocessors communicate with external devices through input and output operations. They can handle various input devices (e.g., keyboards, sensors) and output devices (e.g., displays, printers) by sending and receiving data. I/O operations allow microprocessors to interact with the surrounding environment.

5. Interrupt Handling: Microprocessors support interrupt-driven programming, which allows the processor to respond to external events or internal conditions that require immediate attention. When an interrupt occurs, the microprocessor suspends its current task, saves its state, and transfers control to an interrupt handler routine to handle the event. Interrupt handling is crucial for real-time systems and efficient multitasking.

6. Data Movement: Microprocessors can efficiently move data between different registers, memory locations, and I/O devices. They provide instructions to load data into registers, store data from registers to memory, and transfer data between different I/O devices. Data movement operations are essential for data processing and communication tasks.

7. Instruction Execution: Microprocessors execute instructions stored in memory. These instructions can range from simple arithmetic operations to complex control flow statements. The microprocessor fetches the instructions, decodes them to determine the operation to be performed, and executes them using the appropriate data and control signals.

8. System Control: Microprocessors often include features for system-level control, such as power management, clock control, and bus arbitration. These functions enable the microprocessor to manage its own power consumption, coordinate timing signals, and prioritize access to shared system resources.

It's worth noting that these categories are not mutually exclusive, and many microprocessors perform multiple functions simultaneously. The specific capabilities and features of a microprocessor can vary depending on the architecture and design choices made by the manufacturer.

2. Discuss how data and address busses of 8086 processor are multiplexed and how they are demultiplexed.

Answer:
The Intel 8086 microprocessor, which was introduced in 1978, featured a multiplexed data and address bus. This means that a single set of lines was used to carry both address and data information. Let's discuss how the data and address buses of the 8086 processor are multiplexed and how they are demultiplexed.

1. Multiplexing the Buses:
In the 8086 processor, the multiplexing of the data and address buses is achieved by time-division multiplexing. The 8086 has a 20-bit address bus and a 16-bit data bus. However, the 20 address lines and 16 data lines are combined into a single 16-bit bi-directional bus, called the AD bus.

During the first clock cycle of a bus operation, the AD bus carries the lower 8 bits of the address (A0-A7) and the 16 data lines carry the data to be written or read. This cycle is known as the address phase. During the next clock cycle, the AD bus carries the upper 8 bits of the address (A8-A15) while the data lines remain tri-stated (high-impedance state). This cycle is called the data phase.

2. Demultiplexing the Buses:
To demultiplex the AD bus back into separate address and data buses, external circuitry is required. This circuitry is typically provided by the support chips or external logic connected to the 8086 processor. The demultiplexing process occurs as follows:

- During the address phase, the lower 8 bits of the address (A0-A7) are latched by the external circuitry. This allows the address bus to carry the upper 8 bits of the address (A8-A15) during the data phase.

- Simultaneously, during the address phase, the external circuitry activates the appropriate memory or I/O devices based on the lower 8 bits of the address.

- During the data phase, the external circuitry presents the 16-bit data on the data bus to the memory or I/O devices for writing or reading.

By separating the address and data information through demultiplexing, the external circuitry can properly interpret and handle the address and data signals independently.

It's important to note that the demultiplexing of the buses in the 8086 processor requires additional external components and circuitry. These components, such as address latches and buffers, are responsible for splitting the multiplexed bus into separate address and data buses, ensuring correct communication with memory and I/O devices.

3. Discuss how byte and word data are read from an odd address and even address of an 8086 processor.

Answer:

In the Intel 8086 processor, data can be accessed at both even and odd addresses. However, the processor has a natural alignment preference for accessing data in even address boundaries. Let's discuss how byte and word data are read from odd and even addresses in the 8086 processor:

1. Reading Byte Data:
When reading byte data from an even address (e.g., 0x2000), the 8086 processor fetches the byte from that address directly. For example, if you want to read the byte at address 0x2000, the processor reads it from that location without any additional considerations.

When reading byte data from an odd address (e.g., 0x2001), the 8086 processor follows a specific process called byte alignment. The processor performs the following steps:

a. Fetches the byte from the odd address.
b. Fills the lower byte of a 16-bit register (AX, BX, CX, or DX) with the fetched byte.
c. Fills the upper byte of the register with the byte from the next even address (e.g., 0x2002).

This byte alignment process allows the processor to read a byte from an odd address and utilize the full width of the registers for subsequent operations. The upper byte of the register is filled with the byte from the next even address to ensure proper alignment for word operations.

2. Reading Word Data:
The 8086 processor treats word data (16 bits) differently based on the address's parity (even or odd):

- Even Address: When reading a word from an even address (e.g., 0x2000), the 8086 processor fetches the lower byte of the word from that address and the upper byte from the next even address (e.g., 0x2002). It combines these two bytes to form a 16-bit word.

- Odd Address: When reading a word from an odd address (e.g., 0x2001), the processor performs byte alignment. It fetches the lower byte from the odd address and the upper byte from the next even address (e.g., 0x2002). It then combines these two bytes to form a 16-bit word.

In both cases, the 8086 processor uses the byte alignment technique to ensure proper alignment and utilization of the 16-bit registers.

It's important to note that byte alignment and word access from odd addresses incur a performance penalty due to additional memory accesses required. Thus, it is generally recommended to align data structures and code on even addresses to maximize performance in 8086-based systems.

4. Discuss the internal organization of an 8086 microprocessor.

Answer:
The Intel 8086 microprocessor is a 16-bit processor that features a complex internal organization. Let's discuss the key components and their organization within the 8086 microprocessor:

1. Bus Interface Unit (BIU):
The Bus Interface Unit is responsible for managing external communication and memory access. It consists of the instruction queue, instruction prefetching logic, and the Segment Register Unit. The BIU fetches instructions from memory, decodes them, and prepares them for execution.

2. Execution Unit (EU):
The Execution Unit performs the actual execution of instructions. It includes the Arithmetic and Logic Unit (ALU), general-purpose registers, and control circuitry. The EU carries out arithmetic and logical operations, controls the flow of instructions, and handles interrupts and exceptions.

3. General-Purpose Registers:
The 8086 microprocessor has a set of general-purpose registers, each 16 bits wide. These registers include:

- AX, BX, CX, and DX: These are the primary registers used for arithmetic, data storage, and addressing.
- BP and SP: The Base Pointer and Stack Pointer registers, respectively, are used for stack operations and accessing data on the stack.
- SI and DI: The Source Index and Destination Index registers are used for string manipulation and memory copying operations.
- IP: The Instruction Pointer register stores the memory address of the next instruction to be executed.

4. Segment Registers:
The 8086 processor employs a segmented memory model. It uses segment registers to access different segments of memory. These include:

- CS: The Code Segment register holds the base address of the code segment.
- DS: The Data Segment register points to the base address of the data segment.
- SS: The Stack Segment register points to the base address of the stack segment.
- ES: The Extra Segment register is an additional data segment register used for string operations or additional data segments.

5. Control and Status Registers:

The 8086 microprocessor includes several control and status registers that facilitate its operation. These registers include:

- Flags Register: The Flags register (also known as the Status Register) contains various status bits that reflect the outcome of arithmetic and logical operations, control flow, and condition testing.
- Control Registers: The 8086 processor has control registers that handle system-level control functions, such as the control of interrupts, memory segmentation, and memory protection.

6. Address and Data Buses:
The 8086 microprocessor features a 20-bit address bus and a 16-bit data bus. The address bus is responsible for transmitting memory addresses and I/O locations, while the data bus carries data between the processor and memory or I/O devices.

These components and their organization within the 8086 microprocessor collectively enable it to fetch, decode, and execute instructions, perform arithmetic and logical operations, manage memory access, handle interrupts, and control the flow of data and instructions.

5. Discuss memory segment and different segment registers of 8086 processor.

Answer:
In the Intel 8086 microprocessor, memory is organized using a segmented memory model. The memory is divided into segments, each of which can hold up to 64KB of data. The segment registers in the 8086 processor are used to access these segments. Let's discuss the memory segments and the different segment registers in the 8086 processor:

1. Memory Segments:
In the 8086 processor, memory is divided into the following segments:

- Code Segment (CS): The Code Segment holds the program instructions. It is used in conjunction with the Instruction Pointer (IP) to determine the memory address of the next instruction to be executed.

- Data Segment (DS): The Data Segment stores data used by the program, including variables and arrays.

- Stack Segment (SS): The Stack Segment is used to store the program's stack, which holds temporary data, subroutine return addresses, and local variables.

- Extra Segment (ES): The Extra Segment is an additional data segment that can be used for various purposes, such as storing additional data or facilitating string operations.

2. Segment Registers:
The 8086 processor has four segment registers, each 16 bits wide. These segment registers are used to access different segments of memory. The segment registers in the 8086 processor are as follows:

- Code Segment Register (CS): It points to the base address of the current code segment. It is used in combination with the Instruction Pointer (IP) to fetch instructions.

- Data Segment Register (DS): It points to the base address of the current data segment. It is used to access data items and variables.

- Stack Segment Register (SS): It points to the base address of the current stack segment. It is used for stack operations, such as pushing and popping data.

- Extra Segment Register (ES): It points to the base address of the extra segment. It is an additional data segment that can be used for various purposes.

These segment registers contain a 16-bit selector value that represents the starting address of the segment. The effective memory address is obtained by combining the segment value with a 16-bit offset value obtained from other registers or instructions.

When accessing memory, the physical address is calculated using the formula:
Physical Address = Segment Base Address + Offset

The segment registers and their associated offsets allow the processor to access different segments of memory and provide memory addressing beyond the limitations of a single 64KB segment.

It's important to note that the memory segmentation used in the 8086 processor has limitations and can be complex to work with. Subsequent processor generations, such as the 80386 and beyond, introduced a more advanced and flexible memory model called protected mode, which eliminated the limitations of the segmented memory model.

6. Discuss with example how a physical address is generated by 8086 processor to access a memory segment.

Answer:

To generate a physical address in the Intel 8086 processor, the segment registers and offset values are combined using a specific formula. Let's go through an example to illustrate how a physical address is generated to access a memory segment:

Let's assume we have the following values:

- Code Segment (CS): 0x1000
- Offset (IP): 0x0050

We want to calculate the physical address to access a memory location within the code segment.

1. Load the Code Segment (CS):
The content of the Code Segment register (CS) is loaded with the value 0x1000, which represents the base address of the code segment.

2. Load the Offset (IP):
The offset value (IP) is loaded with the value 0x0050, representing the distance from the base address of the code segment to the desired memory location within the segment.

3. Calculate the Physical Address:
To generate the physical address, the 8086 processor performs the following calculation:

Physical Address = (Segment Base Address × 16) + Offset

In our example:

Physical Address = (0x1000 × 16) + 0x0050
= 0x10000 + 0x0050
= 0x10050

The resulting physical address is 0x10050, which points to the memory location within the code segment where the desired data or instruction resides.

It's important to note that the segment registers and offset values can vary depending on the context and instructions being executed. The example above demonstrates a simple scenario, but real-world programs may involve more complex memory access patterns and calculations.

Additionally, it's worth mentioning that the segmented memory model used in the 8086 processor has limitations and complexities. It was later improved upon in subsequent processor generations, such as the 80386, with the introduction of protected mode and a flat memory model, which offered more flexible and efficient memory addressing.

7. Briefly discuss: 1) Assembly directives 2) DOS and BIOS interrupt 3) Program Segment Prefix

Answer: 1) Assembly Directives:
Assembly directives, also known as pseudo-ops, are instructions in assembly language that provide guidance to the assembler rather than generating machine code. They are used to define constants, reserve memory, specify data types, control the assembly process, and perform other tasks related to code generation. Assembly directives are specific to each assembler and provide a way to enhance code readability and manage various aspects of the assembly process.

Some common assembly directives include:

- .data: Used to define data variables and constants.
- .code: Marks the beginning of the code section.
- .equ: Defines a symbol as a constant value.
- .model: Specifies the memory model for the program.
- .include: Includes an external file in the assembly code.
- .if, .else, .endif: Conditional assembly directives for conditional execution.

2) DOS and BIOS Interrupts:
DOS (Disk Operating System) and BIOS (Basic Input/Output System) interrupts are software interrupt services provided by the operating system and the system BIOS, respectively. These interrupts allow assembly language programmers to access various system functions and services without having to write low-level hardware-specific code.

DOS Interrupts: DOS interrupts provide a set of services to interact with the operating system. They can perform tasks such as file I/O operations, console input/output, process management, and more. DOS interrupts are invoked by specifying the interrupt number (in the AH register) and passing any required parameters in other registers.

BIOS Interrupts: BIOS interrupts provide access to low-level system functions and services, primarily related to hardware operations. These interrupts allow programmers to perform tasks such as keyboard input, video output, disk operations, system initialization, and other system-level functions. Similar to DOS interrupts, BIOS interrupts are invoked by specifying the interrupt number (in the AH register) and passing any necessary parameters in other registers.

3) Program Segment Prefix (PSP):
The Program Segment Prefix is a data structure used by DOS operating systems to store information about a running program. It resides at the

beginning of the program's memory segment and contains various control and status information that DOS uses to manage and communicate with the program.

The PSP holds important information such as the command line arguments, file handles, environment variables, the program's return code, and other system-related data. The PSP is automatically created and initialized by the operating system when a program is launched.

The PSP is typically accessed by assembly language programmers through the segment register DS, which points to the PSP. By manipulating the data in the PSP, programmers can access command-line arguments, modify environment variables, or retrieve other information relevant to the program's execution.

Understanding the PSP structure and utilizing its information can be crucial when developing assembly programs that interact with the operating system and require access to system-level data.

CT-2

Discuss:

Storing a string

Load a string

Scan a string

Compare string

Storing a String:

To store a string in assembly language, you need to allocate memory to hold the characters of the string and then copy the characters into that memory space. The steps involved in storing a string are as follows:

1. Allocate Memory: Reserve memory space using an appropriate directive (e.g., .data or .bss) to define a variable with enough space to hold the string. For example:

```assembly
   myString db "Hello, World!", 0    ; Define a byte array to store the string
```

In this example, `myString` is a label representing the starting address of the string, `db` indicates that it is a byte array, and the string is enclosed in double quotes. The `0` at the end is a null terminator used to mark the end of the string.

2. Load the String:

To load the string into a register or use it in operations, you can load the address of the string into a register using the appropriate addressing mode. For example:

```assembly
```

```
    mov si, offset myString    ; Load the offset of myString into SI register
```

In this example, the `offset` operator retrieves the offset (memory address) of the `myString` label, and it is loaded into the SI register.

Scan a String:

Scanning a string involves reading and processing each character in the string. The steps involved in scanning a string are as follows:

1. Set Up a Loop: Use a loop construct, such as `loop` or `jmp`, to iterate through each character of the string.

2. Access the Characters: Use an addressing mode, such as indirect addressing using registers, to access the characters of the string. For example:

```assembly
    mov al, [si]    ; Load the character at the address stored in SI register into AL
```

In this example, the character at the memory location pointed by the SI register is loaded into the AL register.

3. Process the Character: Perform the desired operations or comparisons on the character. For example, you can check if the character is a specific value, count occurrences, or perform calculations based on the character's value.

4. Update the Pointer: Increment the pointer to move to the next character in the string. For example:

```assembly
   inc si    ; Increment SI register to point to the next character in the string
```

In this example, the SI register is incremented to move to the next memory location.

Compare Strings:

Comparing strings involves checking for equality or determining their relative order. The steps involved in comparing strings are as follows:

1. Set Up Pointers: Set up two pointers, each pointing to the strings you want to compare.

2. Access the Characters: Use an addressing mode to access the characters of the strings using the pointers. For example:

```assembly
   mov al, [si]    ; Load the character from the first string into AL
   mov dl, [di]    ; Load the character from the second string into DL
```

In this example, the characters from the first and second strings are loaded into the AL and DL registers, respectively.

3. Compare the Characters: Perform a comparison operation to compare the characters. Common comparison instructions include `cmp`, `je`, `jne`, etc. For example:

```assembly
   cmp al, dl    ; Compare the characters in AL and DL
   je equal      ; Jump to "equal" label if the characters are equal
   jne not_equal ; Jump to "not_equal" label if the characters are not equal
```

In this example, the `cmp` instruction compares the characters in AL and DL registers. Depending on the result of the comparison, the program can take different actions, such as branching to different labels.

4. Repeat the Process: Repeat steps 2 and 3 until you have compared all the characters in the strings or until a difference is found.

These are the basic steps involved in storing, loading, scanning, and comparing strings in assembly language. The exact implementation may vary depending on the specific assembly language and assembler being used.