

MPCA unit-3

I/O Devices and Interface Units

***Human:** There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.*

The I/O Bus consists of data lines, address lines and control lines.

The I/O bus from the processor is attached to all peripherals interface.

To communicate with a particular device, the processor places a device address on address lines.

explain this to me

Assistant: The passage you provided explains the concept of interface units and how input/output (I/O) communication between the central processing unit (CPU) and peripheral devices (such as printers, keyboards, etc.) is facilitated.

Here's a breakdown of the key points:

1. **Interface units:** These are special hardware components that act as intermediaries between the CPU and peripheral devices. Their primary function is to supervise and synchronize all data transfers between the CPU and peripherals during input and output operations.
2. **I/O Bus:** The I/O bus is a set of wires or lines that carry data, addresses, and control signals between the CPU and peripheral devices. It consists of:
 - **Data lines:** Used to transfer data between the CPU and peripherals.

- **Address lines:** Used to specify the address of the peripheral device that the CPU wants to communicate with.
 - **Control lines:** Used to carry control signals that coordinate the data transfer process.
3. **Connecting peripheral interfaces:** The I/O bus from the processor (CPU) is physically attached to the interface units of all peripheral devices.
 4. **Addressing peripherals:** To communicate with a specific peripheral device, the CPU places the device's address on the address lines of the I/O bus. This address identifies the target device that the CPU wants to exchange data with.

In summary, the CPU communicates with peripheral devices through the I/O bus, which consists of data, address, and control lines. The interface units connected to the I/O bus manage the data transfer between the CPU and the respective peripheral devices. The CPU selects the target peripheral by placing its address on the address lines of the I/O bus, allowing the CPU to send data to or receive data from that specific device.

This mechanism enables the CPU to interact with multiple peripheral devices by addressing them individually and facilitating the exchange of data and control signals through the I/O bus and interface units.

***Human:** Accessing I/O Devices - Memory Mapped Device Interface*

Accessing I/O Devices - I/O Mapped device interface

explain these concepts

Assistant: Sure, I'll explain the two concepts you mentioned: Memory Mapped Device Interface and I/O Mapped Device Interface.

1. **Memory Mapped Device Interface:**

In this approach, the I/O devices are treated as if they are part of the computer's memory address space. Specific memory addresses are

reserved for each I/O device, and these addresses are used to access the device registers or ports.

When the CPU needs to communicate with an I/O device, it performs memory read or write operations to the memory addresses associated with that device. These memory addresses are mapped to the device's registers or ports, allowing the CPU to read data from or write data to the device.

The advantage of this approach is that the CPU can use the same instructions (load and store) that it uses for accessing memory to communicate with I/O devices, simplifying the programming model. Additionally, it allows for faster data transfer rates since memory access is generally faster than separate I/O instructions.

2. I/O Mapped Device Interface:

In this approach, the I/O devices are accessed through a separate I/O address space, distinct from the memory address space. Instead of using memory addresses, the CPU uses specific I/O addresses or port numbers to communicate with I/O devices.

The CPU uses dedicated I/O instructions (like IN and OUT in x86 architecture) to transfer data between the CPU and the I/O devices. These instructions specify the I/O address or port number of the target device, along with the data to be transferred.

The advantage of this approach is that it separates the memory address space from the I/O address space, providing better memory protection and isolation. It also allows for a more straightforward allocation of I/O addresses since they don't need to be mapped to the memory address space.

In modern systems, both Memory Mapped and I/O Mapped interfaces are used, depending on the specific requirements and design of the system. Some devices may use Memory Mapped interfaces, while others may use I/O Mapped interfaces. The choice depends on factors such as

performance, compatibility, and the capabilities of the specific hardware components involved.

It's important to note that these interfaces are implemented at the hardware level and are usually abstracted by device drivers and operating system APIs, simplifying the programming model for application developers.

***Human:** Accessing I/O Devices - Data Transfer Techniques*

Data transfer to and from the peripherals may be done in

any of these ways:

1. Programmed I/O: CPU executes a program and that transfers data between I/O device and Memory

a. Synchronous : Fixed rate of transfer

b. Asynchronous : Handshaking – polling for sending /

receiving the data

c. Interrupt- Driven :(next slide)

2. Direct memory access(DMA): An external controller directly transfers data between I/O device and Memory without CPU intervention.

explain this concept

Assistant: The passage you provided explains the different techniques used for transferring data between input/output (I/O) devices and the computer's memory. These techniques determine how the CPU and other hardware components coordinate and manage the data transfer process. Let me explain each technique:

1. **Programmed I/O:**

In this technique, the CPU executes a program (a set of instructions) to transfer data between an I/O device and the memory. The CPU is

directly involved in the data transfer process, and it follows a specific sequence of instructions to read data from or write data to the I/O device.

There are three subcategories of Programmed I/O:

- a. **Synchronous:** In this mode, the data transfer occurs at a fixed rate. The CPU and the I/O device are synchronized, and the transfer takes place at predetermined intervals.
- b. **Asynchronous:** In this mode, the data transfer is not synchronized, and it occurs at irregular intervals. The CPU continuously checks (polls) the I/O device to see if it is ready to send or receive data. This process is called handshaking.
- c. **Interrupt-Driven:** In this mode, the I/O device sends an interrupt signal to the CPU when it is ready to transfer data. The CPU temporarily suspends its current task and handles the I/O operation based on the interrupt.

2. **Direct Memory Access (DMA):**

In the DMA technique, a separate hardware component called a DMA controller is used to transfer data between an I/O device and the memory. The CPU is not directly involved in the data transfer process.

The CPU initiates the DMA transfer by providing the DMA controller with the source and destination addresses, as well as the amount of data to be transferred. Once the DMA transfer is set up, the DMA controller directly transfers data between the I/O device and the memory without involving the CPU.

The advantage of DMA is that it offloads the CPU from the time-consuming task of managing data transfers, allowing the CPU to perform other tasks concurrently. This improves overall system performance and efficiency, especially for large data transfers.

In summary, Programmed I/O involves the CPU executing a program to manage data transfers, while DMA uses a dedicated hardware controller to transfer data directly between I/O devices and memory without CPU intervention. Programmed I/O can be further categorized into synchronous, asynchronous, and interrupt-driven modes, depending on how the data transfer is coordinated between the CPU and the I/O device.

***Human:** Accessing I/O Devices – Daisy Chain Technique*

In Daisy chain technique, INTR line is common to all the devices.

INTA line is connected in a daisy chain fashion [as shown].

This allows to propagate serially through the devices.

A device when it receives INTA, passes the signal to the next device only if it has not interrupted.

Else, it stops the propagation of INTA and puts the identifying code on the data bus.

Thus the device that is electrically closest to the CPU will have the highest priority.

explain this with a diagram

Assistant: The daisy chain technique is a method used to prioritize and identify the source of interrupts when multiple I/O devices are connected to a single interrupt line (INTR) on the CPU. It involves chaining the interrupt acknowledge (INTA) lines of the devices in a sequential manner, allowing the CPU to determine which device generated the interrupt.

The process works as follows:

1. All devices are connected to a common INTR line, which they use to signal an interrupt to the CPU.
2. The INTA line is connected in a daisy chain fashion, starting from the CPU and propagating through each device sequentially.

3. When an I/O device generates an interrupt, it raises the INTR line, and the CPU responds by asserting the INTA line.
4. The INTA signal propagates serially through the devices.
5. Each device checks if it is the source of the interrupt. If not, it passes the INTA signal to the next device in the chain.
6. If a device is the source of the interrupt, it stops the propagation of the INTA signal and places an identifying code (vector number) on the data bus.
7. The CPU reads the vector number from the data bus, which helps it determine the source of the interrupt and execute the corresponding interrupt service routine (ISR).
8. The device that is electrically closest to the CPU in the daisy chain has the highest priority for interrupt handling.

The daisy chain technique ensures that only one device can claim the interrupt at a time, preventing conflicts and ambiguity. However, it has the limitation that the devices are prioritized based on their physical position in the chain, which may not always align with the desired priority order.

In modern systems, more advanced interrupt management techniques, such as programmable interrupt controllers (PICs) or advanced programmable interrupt controllers (APICs), are commonly used to handle interrupts from multiple devices with more flexibility and configurability.

%% Human:

Accessing I/O Devices – Interrupt Handling Schemes

\ Non-Nested Interrupt Handler

Various stages in a NNIH –

Enable Interrupts

spsr ← spsr_{interrupt request mode}

pc is restored

explain this concept %%

Assistant: This diagram illustrates the concept of a Non-Nested Interrupt Handler (NNIH) and the various stages involved in handling an interrupt in this scheme.

A Non-Nested Interrupt Handler is a simple interrupt handling mechanism where higher priority interrupts are not allowed to interrupt the execution of a lower priority interrupt service routine (ISR). In other words, interrupts are not nested or processed recursively.

The stages in the NNIH process are as follows:

1. **Disable interrupts:** When an interrupt occurs, the first step is to disable further interrupts from being serviced. This is done to prevent interrupt nesting and ensure that the current interrupt is handled without preemption. The program counter (pc) is loaded with the vector table entry corresponding to the interrupt, and the current processor status register (cpsr) is saved in a special register (spsr_{mode}) specific to the interrupt mode.
2. **Save context:** The processor saves the current execution context, which includes the register values and other relevant information, so that it can resume execution from the same state after the interrupt is serviced.
3. **Interrupt handler:** The interrupt handler, which is a routine or a set of instructions, is executed to handle the specific interrupt that occurred. This handler determines the source of the interrupt and performs the necessary operations to service the interrupt.
4. **Service interrupt routine:** The interrupt service routine (ISR) is invoked to perform the actual processing required to handle the interrupt. This routine may involve interacting with the hardware device that generated the interrupt, reading or writing data, or performing other necessary operations.
5. **Restore context:** After the interrupt service routine completes, the processor restores the previously saved execution context. This includes restoring the register values and other relevant information to the state they were in before the interrupt occurred.

6. Enable interrupts: The last step is to re-enable interrupts by loading the program counter (pc) with a value of 0x4, which is a special value that allows interrupts to be serviced again. The cpsr is also restored to the value it had before the interrupt, which is stored in the spsr_{mode} register.

After these steps, the processor returns to the task it was executing before the interrupt occurred, resuming normal execution.

It's important to note that in a Non-Nested Interrupt Handler, if a higher priority interrupt occurs while an ISR is executing, it will be ignored or postponed until the current ISR completes and interrupts are re-enabled. This simplifies the interrupt handling process but may introduce latency for higher priority interrupts.

In more advanced systems, Nested Interrupt Handlers are often used to allow higher priority interrupts to preempt lower priority interrupts, providing better responsiveness and flexibility in handling interrupts of varying priorities.

***Human:** could u differentiate between Non-Nested Interrupt Handler and Daisy Chain Technique*

Assistant: Certainly, I'll differentiate between the Non-Nested Interrupt Handler and the Daisy Chain Technique:

Non-Nested Interrupt Handler (NNIH): The Non-Nested Interrupt Handler is a method for handling interrupts in a processor or system. It focuses on the sequence of steps involved in servicing an interrupt and preventing interrupt nesting (higher priority interrupts interrupting lower priority interrupts). The key aspects of NNIH are:

1. It's a software-based approach for managing interrupts within the processor or system.
2. It disables further interrupts when an interrupt occurs to prevent nesting.

3. It saves the current execution context, executes the interrupt handler and service routine, and then restores the context.
4. It re-enables interrupts after servicing the interrupt.
5. Higher priority interrupts are not allowed to preempt the execution of a lower priority interrupt service routine.

Daisy Chain Technique: The Daisy Chain Technique is a hardware-based method for prioritizing and identifying the source of interrupts when multiple I/O devices are connected to a single interrupt line on the CPU. It involves physically chaining the interrupt acknowledge (INTA) lines of the devices in a sequential manner. The key aspects of the Daisy Chain Technique are:

1. It's a hardware-based approach for managing interrupts from multiple I/O devices.
2. All devices are connected to a common interrupt line (INTR) to signal interrupts to the CPU.
3. The interrupt acknowledge (INTA) lines of the devices are chained or connected in a sequential manner.
4. When an interrupt occurs, the CPU asserts the INTA line, which propagates through the devices.
5. The device that generated the interrupt stops the propagation of the INTA signal and places an identifying code (vector number) on the data bus.
6. The device closest to the CPU in the chain has the highest priority for interrupt handling.

In summary, the Non-Nested Interrupt Handler is a software-based approach that focuses on the sequence of steps for servicing an interrupt and preventing interrupt nesting within a processor or system. The Daisy Chain Technique, on the other hand, is a hardware-based approach that focuses on prioritizing and identifying the source of interrupts from multiple I/O devices connected to a single interrupt line on the CPU.

The concept you've described is known as "Cycle Stealing" or "Cycle Steal" technique, which is a method used in computer systems to allow

external devices or controllers to access memory or the system bus without interfering with the CPU's normal operation.

DMA:

Here's an explanation of the key points:

1. **Purpose:** Cycle Stealing is a way to allow external devices, such as I/O controllers or peripherals, to access the system's memory or bus without interrupting the CPU's execution. This is achieved by taking advantage of the CPU's idle cycles or periods when it is not actively using the memory or bus.
2. **Similar to DMA:** Cycle Stealing is similar to Direct Memory Access (DMA) in that it allows devices other than the CPU to access memory directly. However, unlike DMA, where a separate DMA controller manages the memory transfers, Cycle Stealing relies on the device itself to "steal" cycles from the CPU when it is not using the memory or bus.
3. **Exploiting CPU/Bus Timings:** Cycle Stealing works by exploiting specific timings and cycles of the CPU or bus. When the CPU is not actively accessing memory or the bus (e.g., during instruction fetches or execution), external devices can take control of the memory or bus and perform their own read or write operations.
4. **Dual-Port RAM Alternative:** Cycle Stealing provides a cost-effective alternative to dual-port RAM, which is a type of memory that allows simultaneous access by two devices (e.g., CPU and I/O controller). By using Cycle Stealing, systems can achieve similar functionality without the need for expensive high-speed dual-port RAM.
5. **CPU Halting or Full-Speed Operation:** In some implementations, the CPU may need to be halted or temporarily paused during the cycle stealing process to prevent conflicts or data corruption. However, in more advanced systems, cycle stealing can be designed to occur seamlessly without halting the CPU, allowing it to run at full

speed while external devices access memory during the CPU's idle cycles.

6. **CPU/Bus Conflict Avoidance:** To ensure data integrity and prevent conflicts, cycle stealing mechanisms must carefully manage the timing of memory or bus accesses by external devices. The external device must complete its operations before the CPU requires access to the same memory or bus resources to avoid any potential conflicts.

Cycle Stealing is a technique that aims to improve system efficiency by allowing external devices to access memory or the bus without significantly impacting the CPU's performance. It takes advantage of the CPU's idle cycles and carefully manages the timings to avoid conflicts, providing a cost-effective alternative to dedicated DMA controllers or dual-port RAM solutions.

Peripheral Component Interconnect : PCI-Bus Architecture :

The concept you're describing is the Peripheral Component Interconnect (PCI) bus architecture, which is a widely used standard for connecting peripheral devices to a computer system. Here's an explanation of the key features and operations:

1. **Bus Width:** The PCI bus is a 32-bit bus, meaning it can transfer 32 bits (4 bytes) of data at a time.
2. **Transfer Rate:** The PCI bus has a maximum transfer rate of 133 MB/s (megabytes per second), which allows for relatively fast data transfer between the CPU and peripheral devices.
3. **Device Initiator:** Any PCI device connected to the bus can initiate a transaction, such as a data transfer or configuration operation.
4. **Bus Arbitration:** Before a device can initiate a transaction, it must request permission from the PCI bus arbiter, which is typically implemented on the motherboard.

5. **Bus Arbiter:** The PCI bus arbiter is responsible for granting permission to one of the requesting devices at a time. It ensures that only one device can access the bus at any given time, avoiding conflicts and data corruption.
6. **Transaction Initiation:** When a device is granted permission by the arbiter, it can initiate a transaction by broadcasting a 32-bit address and a 4-bit command code on the bus.
7. **Address Phase:** After broadcasting the address and command code, the initiator device waits for a target device to respond.
8. **Target Device Selection:** All devices connected to the PCI bus examine the broadcasted address. The device that recognizes the address as its own responds a few cycles later, indicating that it is the target for the transaction.
9. **Data Transfer:** Once the target device has responded, the initiator and target devices can proceed with the data transfer or configuration operation, depending on the command issued.

The PCI bus architecture allows for efficient communication between the CPU and peripheral devices by providing a standardized and high-performance interconnect. Its key features include a 32-bit bus width, a maximum transfer rate of 133 MB/s, the ability for any device to initiate transactions, bus arbitration to manage access, and a protocol for addressing and communicating with target devices.

This architecture simplified the integration of peripheral devices into computer systems and improved overall system performance compared to older bus standards. It was widely adopted in personal computers and servers until newer standards like PCI Express (PCIe) were introduced to meet the increasing demands for higher data transfer rates and more efficient bus utilization.

SCSI – Bus Architecture :

The Small Computer System Interface (SCSI) is a set of standards and protocols for connecting and transferring data between computers and peripheral devices, particularly storage devices like hard disks, CD-ROMs, and tape drives.

Here's an explanation of the key points you mentioned:

1. **Origins:** SCSI was derived from the Shugart Associates System Interface (SASI), which was an earlier interface standard used to connect hard disks to computer systems.
2. **Purpose:** SCSI was designed to provide a bridge between low-level interfaces of hard disks and the host computer, enabling efficient communication and data transfer.
3. **Standard Interface:** SCSI defines a standard interface and communication protocol, allowing various types of peripheral devices from different manufacturers to be connected to a computer system.
4. **Performance:** One of the main goals of SCSI was to increase performance and deliver faster data transfer rates compared to other interfaces at the time.
5. **Device Addressing:** A SCSI bus can address up to 8 devices, with device IDs ranging from 0 to 7. This allows for the connection of multiple devices to the same SCSI bus.
6. **Expansion Capabilities:** SCSI provides larger expansion capabilities, enabling the connection of various peripheral devices such as CD-ROMs, scanners, and other storage devices.
7. **Client-Server Architecture:** SCSI is based on a client-server architecture model.
 - **Initiators (Clients):** Devices that create (initiate) and send SCSI commands to the targets. For example, a computer's SCSI controller acts as an initiator.

- **Targets (Servers):** Devices that receive and carry out the requested tasks from the initiators. Targets are typically collections of logical units (e.g., hard disk drives, CD-ROM drives) that perform the requested operations.

8. **Bus Architecture:** Early versions of SCSI assumed a bus-based architecture, where devices were connected to a shared bus (similar to a parallel bus structure).

The SCSI interface provided a standardized way to connect and communicate with various peripheral devices, particularly storage devices. It offered improved performance, expandability, and compatibility compared to earlier interfaces. SCSI played a crucial role in the evolution of computer storage systems and peripherals until newer interfaces like Serial Attached SCSI (SAS) and Serial ATA (SATA) were introduced to meet the increasing demands for higher data transfer rates and more efficient communication protocols.

SCSI Bus Operation in phases:

This diagram illustrates the different phases involved in the operation of the SCSI (Small Computer System Interface) bus when an initiator device (such as a computer's SCSI controller) needs to communicate with a target device (such as a hard disk drive or CD-ROM drive).

The SCSI bus operation is divided into the following phases:

1. **BUS free:** This phase indicates that the SCSI bus is currently idle and available for use by any initiator device that needs to communicate with a target device.
2. **Arbitration:** When an initiator device wants to use the SCSI bus, it must first go through an arbitration process. During this phase, multiple initiators can compete for control of the bus by asserting their respective arbitration signals. The initiator with the highest priority (based on a predefined priority scheme) wins the arbitration and gains control of the bus.

3. **Selection:** After winning the arbitration, the initiator selects the target device it wants to communicate with by sending the target's unique SCSI ID over the bus. The target device recognizes its ID and responds, acknowledging the selection.
4. **Message:** Once the target device is selected, the initiator can send a SCSI command or message to the target. This message contains instructions for the target device, such as reading data from a specific location on a hard disk or writing data to a CD-ROM.
5. **Data/Command/Msg Status:** In this phase, the actual data transfer or command execution takes place. Depending on the SCSI command, data may be transferred from the initiator to the target (write operation), from the target to the initiator (read operation), or both ways. Additionally, the target may send status messages back to the initiator regarding the success or failure of the command execution.

The SCSI bus operation follows these phases to ensure proper communication and data transfer between the initiator and target devices. The arbitration phase ensures that only one initiator has control of the bus at a time, preventing conflicts. The selection phase allows the initiator to specify the target device it wants to communicate with, and the message and data transfer phases facilitate the actual exchange of commands and data.

This phased approach to SCSI bus operation provides a structured and organized way for multiple devices to share the same bus and communicate effectively, while also handling potential conflicts and ensuring data integrity.

AMBA (Advanced Microprocessor Bus Architecture) protocol :

The AMBA (Advanced Microprocessor Bus Architecture) protocol is an open standard developed by Arm for on-chip interconnect specifications in System-on-Chip (SoC) designs. It defines a set of interfaces and protocols to facilitate the connection and management of various

functional blocks, such as processors, memory, and peripherals, within an SoC.

The key features of AMBA interfaces are:

1. **IP reuse:** AMBA specifications aim to enable the reuse of intellectual property (IP) blocks, which is essential for reducing development costs and timescales in SoC designs. By adhering to the AMBA standards, IP blocks from different vendors can be easily integrated into an SoC design, promoting IP reuse.
2. **Flexibility:** AMBA interfaces provide flexibility by supporting a wide range of interconnect requirements, including different bus widths, transfer sizes, and topologies. This flexibility allows the AMBA protocols to be used in various SoC designs with varying performance and power requirements.
3. **Wide Adoption:** AMBA protocols have gained widespread adoption within the semiconductor industry, particularly in Arm-based SoC designs. This broad adoption ensures compatibility and interoperability between IP blocks from different vendors, facilitating seamless integration and reducing development efforts.
4. **Compatibility:** AMBA interfaces are designed to be backward-compatible, allowing newer versions to work with older implementations. This compatibility ensures that existing IP blocks can be reused in new SoC designs, reducing the need for significant modifications and minimizing development efforts.
5. **Support:** AMBA protocols are well-documented and supported by Arm and the industry. Arm provides comprehensive documentation, tools, and resources to aid in the development and verification of AMBA-compliant designs. Additionally, the widespread adoption and industry support ensure that developers have access to a large ecosystem of compatible IP blocks and development tools.

The AMBA protocols play a crucial role in enabling right-first-time development of multi-processor SoC designs with large numbers of controllers and peripherals. By providing a standardized and widely adopted interconnect specification, AMBA facilitates the integration of various IP blocks from different vendors, promotes IP reuse, and helps reduce development costs and timescales. Its flexibility, compatibility, and industry support make it a valuable standard for SoC design and development.

APB (Advanced Peripheral Bus) protocol :

The Advanced Peripheral Bus (APB) is one of the key protocols within the AMBA (Advanced Microprocessor Bus Architecture) specification defined by Arm. It is a low-cost and low-power on-chip bus designed for connecting low-bandwidth peripheral devices to the high-performance system backbone in a System-on-Chip (SoC) design.

The APB protocol has the following key features:

1. **Simplicity:** APB is a simple and lightweight protocol, making it suitable for low-performance and low-power peripheral devices that do not require high data transfer rates.
2. **Low Power:** APB is designed to minimize power consumption, which is crucial for power-sensitive applications such as embedded systems and mobile devices.
3. **Bridge-Based Architecture:** APB operates as a separate bus and is typically connected to the high-performance system backbone (such as the AXI or AHB buses) through a bridge component. This bridge acts as an interface between the APB and the higher-performance buses, allowing communication between the peripheral devices and the rest of the system.
4. **Pipelined Operation:** APB supports pipelined operation, allowing new transfer requests to be accepted before the previous transfer

has completed. This pipelining capability can improve the overall system performance by reducing idle cycles.

5. **Peripheral Interface:** APB is primarily used for connecting low-bandwidth peripheral devices such as timers, UARTs, keyboards, and other input/output devices to the SoC system.
6. **Clocking Scheme:** APB can operate with different clocking schemes, including support for clock gating and clock scaling, which further contribute to power efficiency.

The APB protocol is often used in conjunction with other AMBA protocols, such as the Advanced High-performance Bus (AHB) or the Advanced eXtensible Interface (AXI), which provide high-performance system interconnects. In a typical SoC design, the APB bus is connected to the high-performance system bus through a bridge, allowing efficient communication between high-performance components (such as processors and memory) and low-performance peripherals.

By providing a simple and low-power interface for connecting peripheral devices, the APB protocol plays a crucial role in enabling efficient and power-optimized SoC designs, particularly in the embedded and mobile computing domains where power efficiency is a critical consideration.