Course Title: Microprocessor System & Interfacing
Course Code: CPE 403
December 4, 2024

1 Course Overview

This course offers a comprehensive study of microprocessor systems, covering their essential components, architecture, programming, interrupts, and interfacing methods. Through a combination of theory and hands-on experience, students will gain practical skills in designing, implementing, and troubleshooting microprocessor-based applications. Key topics include microprocessor architecture, instruction sets, programming models, memory and I/O interfacing, and peripheral devices, with a strong focus on Intel microprocessors. Additionally, this course delves into advanced interfacing topics such as USB, PCIe, serial communication, Ethernet, and CAN bus. The coursework emphasizes the use of Intel assembly language, equipping students with the foundational skills needed for developing efficient, low-level microprocessor programs.

1.1 Course Learning Objectives (CLO)

By the end of the course a successful student should have the following understanding about the course.

1. Understand the core components and functions of microprocessor systems, including their history and evolution.
2. Describe and analyze the architecture of Intel microprocessors, including the role of registers, ALUs, control units, and other key components.
3. Formulate programming models for Intel microprocessors and recognize how to utilize various registers and instruction sets effectively.
4. Demonstrate the ability to write, execute, and debug assembly language programs using Intel instruction sets for basic to complex tasks.
5. Apply interfacing techniques to connect memory and I/O devices to microprocessors, ensuring efficient data transfer and functionality.
6. Understand and implement interrupt mechanisms, including handling hardware and software interrupts to manage various system events.

7. Interface with peripheral devices such as ADCs, DACs, timers, and counters, enabling microprocessors to interact with external components effectively.

8. Apply communication protocols including USB, PCIe, serial communication, Ethernet, and CAN bus, connecting microprocessors to a range of external systems.

9. Design and construct microprocessor-based systems, integrating multiple components to build a cohesive and functional platform.

10. Diagnose and resolve issues in microprocessor systems, identifying and addressing both hardware and software problems efficiently.

11. Explore advanced interfacing techniques for high-speed and complex communication between microprocessors and other devices, focusing on real-world applications.

12. Design, implement, and evaluate a microprocessor-based application or project that showcases comprehensive interfacing and programming knowledge.

2 Introduction to Microprocessor Systems

In the old days, transistors were made individually and then formed into an electronic circuit with the use of wires and solder. Today transistors are part of an integrated circuit—an entire electronic circuit, including wires, formed on a single "chip", or piece, of special material, usually silicon, as part of a single manufacturing process. Inte- grated circuits were developed by *Jack Kilby* at Texas Instruments, who demonstrated the first one in 1958.

An integrated circuit embodies what is called solid-state technology. In a solid-state device, the electrons travel through solid material with no moving parts—in this case, silicon. They do not travel through a vacuum, as was the case with the old radio vacuum tubes.

A microprocessor is an integrated circuit (IC) that performs the core computations and control functions of a computer by executing instructions from memory. Unlike simpler computing circuits, it has a central processing unit (CPU) embedded on a single chip, incorporating multiple functional units (such as arithmetic logic units, control units, and registers) that handle complex tasks like arithmetic and logical operations, control flows, and data movement.

Microprocessors are the backbone of many modern computing systems, including embedded systems, personal computers, and servers. These chips can execute thousands of instructions per second, driven by advanced features such as:

- **Superscalar Architecture**: Many modern microprocessors use a superscalar architecture, allowing them to execute multiple instructions simultaneously, thus enhancing throughput.
- **Pipelining**: This technique breaks down instruction processing into stages (fetch, decode, execute, etc.) to enable faster processing, as multiple instructions can be at different stages within the processor at the same time.
- **Multithreading**: Some microprocessors employ simultaneous multithreading (SMT) to execute multiple threads concurrently, maximizing core usage and performance.
- **Parallelism**: Modern processors support SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction, Multiple Data) paradigms to handle parallel operations across multiple data points, commonly used in vector processing, multimedia, and scientific computations.
- **Cache Memory**: High-speed memory (L1, L2, and sometimes L3 caches) on-chip helps reduce latency by storing frequently accessed data close to the processor.
- **Branch Prediction and Speculative Execution**: To improve efficiency, advanced microprocessors predict the flow of branches in code and execute instructions ahead of time, discarding the results if predictions are incorrect.
- **Clock Speed and Power Management**: Modern microprocessors balance high clock speeds with power-saving modes (dynamic voltage and frequency scaling) to optimize performance and efficiency for varying workloads.

NOTE: *In embedded systems (CPE 503), the versatility and configurability of microprocessors, especially RISC (Reduced Instruction Set Computing) architectures like ARM (Advanced RISC Machine), enable them to cater to specific tasks in fields such as automotive, IoT, aerospace, and medical devices. Meanwhile, x86 architectures (commonly found in Intel (Integrated Electronics) and AMD (Advanced Micro Devices) processors) dominate general-purpose computing and data centers due to their robust performance in multitasking and support for a large instruction set.*

Microprocessors have revolutionized the way we live and work, and their importance is undeniable.

(i) **Central Processing and Control**: They are the "brains" of computers and embedded systems, handling complex computations, controlling devices, and managing tasks in a wide range of applications, from smartphones to industrial control systems or machines.

(ii) **Miniaturization and Efficiency**: Microprocessors integrate multiple functions into a single chip, which significantly reduces the size, cost, and power requirements of electronic systems.

(iii) **Versatility**: Microprocessors are highly programmable, allowing them to adapt to various applications, from consumer electronics to automotive systems and industrial automation.

(iv) **Advancements in Technology**: As microprocessors improve in performance, speed, and efficiency, they enable advancements in fields like artificial intelligence, machine learning, telecommunications, big data processing, etc.

Microprocessor acts as the brain of the computer (PC) or electronic devices. The primary purpose of a microprocessor is to perform a wide range of computational tasks by executing instructions, enabling various applications. They are critical to modern electronics for below reason:

(i) **Data Processing**: Microprocessors perform arithmetic, logical, and comparison operations on data, essential for computations and decision-making.

(ii) **Control Operations**: They manage and control the operation of other components within a system, orchestrating processes, and maintaining a steady flow of tasks based on instructions received.

(iii) **Input and Output Coordination**: Microprocessors handle data exchange with external peripherals and memory, managing input and output operations crucial for system interaction.

(iv) **Multitasking**: In advanced systems, microprocessors handle multiple tasks simultaneously, often running operating systems, applications, and background services concurrently.

(v) **Signal Processing**: In embedded systems, microprocessors are crucial for real-time signal processing in applications like audio, video, and sensor data analysis.

The architecture of a microprocessor includes several fundamental components and structures, there are discussed below:

1. **Control Unit (CU)**: The control unit manages the execution of instructions by directing the operations of the other components. It decodes instructions, controls data flow, and manages control signals to synchronize all parts of the processor.

2. **Arithmetic Logic Unit (ALU)**: The ALU is responsible for performing arithmetic (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT, etc.). It is the core computational unit within the microprocessor.

3. **Registers**: Registers are small, fast storage locations within the processor used to temporarily hold data and instructions during execution. Common types include:
   (a) **Accumulator**: Holds intermediate results of calculations.
   (b) **Instruction Register (IR)**: Holds the current instruction being executed.
   (c) **Program Counter (PC)**: Points to the next instruction to be executed in memory.
   (d) **Stack Pointer (SP)** and Base Pointer (BP): Used for managing function calls and memory addressing.
4. **Bus Interface**: Buses are data pathways that connect the microprocessor with other components. The primary buses in a microprocessor are:
   (a) **Data Bus**: Transports data between the processor, memory, and peripherals.
   (b) **Address Bus**: Carries memory addresses from the processor to memory or I/O devices.
   (c) **Control Bus**: Sends control signals between the processor and other components to coordinate actions.
5. **Cache Memory**: Modern microprocessors have cache memory, typically organized in multiple levels (L1, L2, sometimes L3), to store frequently accessed data close to the processor, reducing latency and improving performance.
6. **Clock**: The clock provides timing signals to synchronize all actions within the processor. Clock speed, measured in Hertz (Hz), determines how many instructions per second the processor can execute.

Microprocessor architectures are broadly classified into two types based on instruction set design:

(a) **CISC (Complex Instruction Set Computing)**: CISC machines have a large set of instructions with complex, multi-cycle operations. It enables complex tasks to be executed with fewer instructions, though each instruction may take longer. They are common in x86 processors (Intel, AMD).

(b) **RISC (Reduced Instruction Set Computing)**: They utilizes a smaller, simpler set of instructions, optimized for speed and efficiency. Each instruction typically executes in a single clock cycle, leading to faster performance in many applications. They are common in ARM processors used in embedded systems, smartphones, and some servers.

Figure 1: Pictorial diagram of a CPU

2.1 Components of Microprocessor (CPU)

In a computer system, the four primary components: **CPU** (Central Processing Unit), **Memory**, **I/O** (Input/Output), and **Bus Subsystems** work together to execute instructions, manage data, and communicate within the system. The below components form a cohesive unit that enables the microprocessor to handle complex instructions and manage tasks effectively within a wide range of devices, from simple embedded systems to advanced supercomputers.

(i) **CPU (Central Processing Unit)**: The CPU, often called the "brain" of the computer, is responsible for executing instructions and performing calculations. Below are some components of the CPU:

- **Control Unit (CU)**: Directs the operations of the CPU, managing the flow of data between the CPU, memory, and I/O devices.
- **Arithmetic Logic Unit (ALU)**: Performs arithmetic and logical operations on data.
- **Registers**: Small, fast storage locations inside the CPU used to hold data and instructions temporarily during processing.
- **Cache**: A high-speed memory area in the CPU that stores frequently accessed data, improving processing speed.

The CPU fetches instructions from memory, decodes them, executes them, and then stores the results. It manages all processing tasks and

coordinates with other subsystems.

(ii) **Memory**: It stores data, instructions, and intermediate results for quick access by the CPU. Below are the types of memory that the CPU interacts with:

- **RAM (Random Access Memory)**: Volatile memory used for temporarily storing data and instructions that the CPU needs during operation. Data is lost when the power is turned off.
- **ROM (Read-Only Memory)**: Non-volatile memory used to store firmware or system software that does not change frequently, like the BIOS in a computer.
- **Cache Memory**: A small, fast memory located within or close to the CPU. It holds frequently accessed data to reduce the time needed to access main memory.

Memory provides the CPU with quick access to the data and instructions required to perform tasks, ensuring smooth and efficient operation.

(iii) **I/O (Input / Output)** I/O devices allow the computer to communicate with the outside world by providing means for input (data coming into the system) and output (data going out of the system). Below are the types of I/O devices

- **Input Devices**: Include keyboards, mice, scanners, and sensors, which enable users or other systems to send data to the computer.
- **Output Devices**: Include monitors, printers, speakers, and actuators, which allow the computer to convey data to users or other systems.
- **Storage Devices**: Such as hard drives, SSDs, and USB drives, which provide long-term data storage and retrieval.

I/O devices expand the functionality of the computer, allowing it to interact with external systems, peripherals, and users, thus supporting a broad range of applications.

(iv) **Bus subsystems**: The bus is a **communication pathway** used to transfer data between different parts of the computer, like the CPU, memory, and I/O devices. Below are some different types of bus subsystems.

- **Data bus**: Transfers actual data between the CPU, memory, and I/O devices.
- **Address Bus**: Carries the addresses of memory locations or I/O devices that the CPU wants to read from or write to.
- **Control Bus**: Sends control signals from the CPU to other components to manage the operations, such as read/write commands or interrupt signals.

Bus subsystems facilitate data transfer and ensure synchronization among components. They enable the CPU to efficiently access memory and communicate with I/O devices, contributing to the overall performance and reliability of the system.

2.2 Evolution of Microprocessor (8 to 64 bits)

Microprocessor evolution has enabled the development of modern computing, from early calculators and simple embedded devices to powerful desktop computers, mobile devices, and servers that drive today's digital world. This progression not only reflects technological advancements in hardware but also the demands of increasingly complex applications and the evolving needs of users. The future of microprocessors is poised for further innovation. Advancements in semiconductor technology, such as nanotechnology, promise to deliver even more powerful and energy-efficient processors. As we move towards an increasingly interconnected world, microprocessors will continue to play a pivotal role in shaping the future of technology.

Bit Size	**Year Introduced**	**Key Features**	**Example Processors**
4-bit	Early 1970s	Basic processing capabilities, used in calculators and early embedded systems	Intel 4004, Texas Instruments TMS1000
8-bit	1970s	Simple instruction set, limited memory addressing	Intel 8080, Zilog Z80, Motorola 6800
16-bit	Late 1970s - 1980s	Increased memory addressing, support for more complex instructions	Intel 8086, Motorola 68000
32-bit	1980s - 1990s	Improved data handling, multitasking, larger address space	Intel 80386, Motorola 68020, ARM Cortex-A8
64-bit	Late 1990s - Present	Extended address space, advanced computing capabilities, support for modern OS and applications	Intel Pentium 4, AMD Athlon 64, ARM Cortex-A72

Table 1: Evolution of Microprocessors from 4-bit to 64-bit

The evolution of microprocessor bit sizes, typically from 4-bit up to 64-bit architectures

as discussed in the above table, has been driven by the increasing demand for processing power, memory addressing capabilities, and data handling efficiency. Each leap in bit size has enabled new levels of performance, functionality, and application scope. The breakdown of how each step contributed to microprocessor evolution is discussed below:

1. **4-bit Microprocessors**

    - These processors were designed for basic computation tasks and embedded applications, such as calculators and simple digital control systems.
    - The 4-bit architecture limited data processing to very small chunks and could address only a minimal amount of memory. This constrained them to performing basic arithmetic and logical operations.
    - The 4-bit era was foundational, proving that integrated processors could handle computation tasks. It paved the way for more advanced microprocessor designs.

2. **8-bit Microprocessors**

    - They were a major step forward, enabling more complex calculations and broader applications, including personal computing.
    - With 8 bits, microprocessors could handle one byte at a time, providing greater data throughput and memory addressing capabilities than 4-bit designs. They also supported simple operating systems and allowed for primitive multitasking.
    - 8-bit processors were critical in popularizing microcomputers and home computing, such as the Apple II and Commodore 64. They proved that microprocessors could manage more diverse applications, from business software to games.

3. **16-bit Microprocessors**

    - These processors could handle larger data types and address more memory, expanding the possible applications.
    - The 16-bit architecture allowed processors to handle two bytes simultaneously, significantly improving performance. It also enabled more complex operating systems and applications, setting the stage for professional software and graphical interfaces.
    - 16-bit processors were widely adopted in early personal computers (e.g., IBM PC) and embedded systems, marking the transition from simple computing tasks to more sophisticated applications in business, engineering, and software development.

4. **32-bit Microprocessors**

    - The rise of 32-bit processors, such as the Intel 80386 and ARM architecture brought a major leap in processing power and memory addressing, enabling more powerful personal computers and workstations.
    - With 32 bits, processors could address up to 4GB of RAM and handle large data operations more efficiently. This was essential for multitasking operating systems like Windows and Unix, as well as for handling complex graphics and large databases.
    - 32-bit processors became the standard in both consumer and professional computing. They were pivotal for advanced applications like gaming, multimedia, and high-performance computing tasks. Embedded systems also benefited, with 32-bit chips providing the power for automotive, telecommunications, and industrial control applications.

5. **64-bit Microprocessors**

    - 64-bit microprocessors, such as the AMD64 and Intel x86-64, targeting high-end workstations, servers, and later personal computers. They allowed significantly more memory addressing and data processing capabilities.
    - The 64-bit architecture can theoretically address up to 16 exabytes of memory, a capacity far beyond current needs but essential for future-proofing. Additionally, 64-bit processors can handle larger data types natively, improving performance in applications that involve large datasets, complex simulations, and high-end graphics.
    - 64-bit processors have become the standard across computing devices, from desktops and laptops to mobile devices and servers. They enable the processing power needed for modern operating systems, virtualization, AI, and big data analytics, and support a much broader memory space, which is essential for advanced computing applications.

6. **Beyond 64-bit (128-bit and Quantum Computing)**: Although 128-bit processors are theoretically possible, there is limited demand due to the sufficient capabilities of 64-bit processors. However, quantum computing and specialized processors, such as GPUs with hundreds or thousands of cores, are expanding what's possible in computing without relying solely on traditional bit-size evolution. As computing demands grow in fields like artificial intelligence, cryptography, and scientific research, these advanced architectures could mark the next evolution phase beyond the classic bit-size progression.

Microprocessor bit-size evolution has been tied to the expanding requirements of computing applications, from simple arithmetic tasks to the demands of modern data-intensive

applications and AI. This evolution reflects a natural progression to support increasing complexity, processing speed, and memory requirements across a wide variety of industries and use cases.

2.3 Microcontroller vs. Microprocessor

Microcontrollers, also called embedded computers, are the tiny, spe- cialized microprocessors installed in "smart" appliances and auto-mobiles. These microcontrollers enable microwave ovens, ***for example, to store data about how long to cook your potatoes and at what power setting***. Microcontrollers have been used to develop a new universe of experimental electronic appliances—e-pliances. For example, they are behind single-function products such as digital cameras and digital music players, which have been developed into hybrid forms such as gadgets that store photos and videos as well as music. They also help run tiny machines embedded in clothing, jewelry, and such household appliances as refrigerators. In addition, microcontrollers are used in blood-pressure monitors, air bag sensors, gas and chemical sensors for water and air, and vibration sensors.

Aspect	Microcontroller	Microprocessor
Definition	A compact integrated circuit that includes a processor, memory, and I/O peripherals.	A central processing unit (CPU) that requires external memory and I/O peripherals.
Main Usage	Primarily used for dedicated tasks in embedded systems, such as controlling appliances, sensors, and automotive electronics.	Primarily used in general-purpose systems, like computers, where they handle complex tasks and multitasking.
Memory	Built-in RAM, ROM, and sometimes flash memory for storage; memory is often minimal and application-specific.	No internal memory; requires external memory (RAM and ROM) to store data and programs.
I/O Ports	Multiple integrated I/O ports, ADCs, and communication protocols like UART, SPI, I2C for easy interfacing with sensors and actuators.	Limited or no built-in I/O ports; needs external components to manage I/O, increasing system complexity.
Power Consumption	Typically low, as they are optimized for battery-operated, energy-efficient systems.	Generally higher due to external components and their high-speed processing capabilities.
Cost	Usually low-cost, suitable for cost-sensitive applications.	Typically higher cost due to the need for additional components and higher processing power.
Programming Complexity	Often uses simpler, application-specific programming, with real-time operating systems or firmware.	More complex, with a wide variety of operating systems and multitasking capabilities, suitable for applications requiring significant processing power.
Examples	Arduino (ATmega328), STM32, PIC series.	Intel x86, ARM Cortex-A series, AMD Ryzen.
Application Areas	Appliances, IoT devices, robotics, automotive systems, medical devices.	Personal computers, gaming consoles, smartphones, high-performance computing.

Table 2: Comparison between Microcontrollers and Microprocessors

3 Microprocessor Architecture and Operation

This revolve around the fundamental principles that define how central processing units (CPUs) interpret and execute instructions. Microprocessors operate based on a series of

integrated circuits that execute instructions from memory to perform arithmetic, logical, control, and input/output (I/O) operations. Understanding their architecture and operation provides insight into how computers and embedded systems function. In embedded systems devices, the architecture and efficiency of microprocessors play a crucial role, particularly for applications that require low power and high reliability. The continual evolution of microprocessor design, with a focus on power efficiency and performance, underpins much of the progress in modern computing and embedded systems (CPE 503).

3.1 Microprocessor Architecture

It is the fundamental design and structure of a microprocessor. It determines how it processes data and interacts with memory and peripherals. The architecture of a microprocessor significantly influences its performance, power consumption, and applicability for different types of tasks, from embedded systems to high-performance computing. Below are the main components and aspects of the microprocessor:

1. **Arithmetic and Logic Unit (ALU)**: The ALU is the core component responsible for performing arithmetic operations (addition, subtraction, multiplication, division) and logic operations (AND, OR, XOR, NOT). It takes input data from registers, performs the computation, and sends the result back to the registers or memory.

2. **Control Unit (CU)**: The CU coordinates the microprocessor's activities by interpreting instructions from memory and generating the necessary control signals for other parts of the microprocessor. It directs the flow of data within the processor and manages interactions with peripherals. The CU typically involves an instruction decoder to decode the binary instruction code into control signals.

3. **Registers**: Registers are small, fast memory locations within the CPU that hold data temporarily for quick access. General-purpose registers are used for temporary data storage during computation, while special-purpose registers (like the accumulator, program counter, and status register) play specific roles in execution and control. Accumulator (ACC): Holds intermediate results of operations. Program Counter (PC): Keeps track of the address of the next instruction to execute. Stack Pointer (SP): Points to the top of the stack, a specific memory region for temporary data storage. Status/Flags Register: Holds flags to indicate conditions (e.g., zero, carry, overflow) based on the results of operations.

4. **Bus Interface Unit (BIU)**: The BIU manages communication between the CPU and external devices through a set of buses. Below are the detail explanation of BIU used in computer system:

(a) **Data Bus**: The data bus carries the actual data being transferred between the *processor, memory, and peripherals.* It is usually bidirectional, allowing data to flow in both directions (from CPU to memory/peripherals and vice versa). The width (in bits) of the data bus determines the amount of data the processor can handle at once. *For instance, a 32-bit data bus can transfer 32 bits of data simultaneously, which impacts processing speed and efficiency.*

(b) **Address Bus**: The address bus is responsible for carrying the memory addresses from the CPU to other components, allowing the CPU to specify the location in memory for data storage or retrieval. Typically, the address bus is unidirectional, **meaning it only goes from the CPU to memory or peripherals**. The width of the address bus determines the **maximum memory capacity** the processor can access. For instance, a 16-bit address bus can address up to 2^{16} (65,536) memory locations.

(c) **Control Bus**: The control bus **carries control signals** from the CPU to coordinate and manage the operations of the computer, like ***reading or writing to memory, interrupt handling, and power management***. Control signals include **Read, Write, Interrupt, and Reset**, among others. It can be either unidirectional or bidirectional, depending on the control signal direction.

(d) **System Bus**: The system bus is a combination of the *data, address, and control buses*, collectively forming the main pathway for data, address, and control signals to move within the system. It **connects the CPU to the main memory**, facilitating primary data communication.

(e) **Expansion Bus**: The expansion bus allows external peripherals (such as graphics cards, network cards, etc.) to communicate with the CPU and memory, extending the capabilities of the system. Examples include PCI (Peripheral Component Interconnect), USB, and AGP (Accelerated Graphics Port) buses in PCs. ***The expansion bus often operates at a slower speed than the system bus, as it typically handles I/O operations rather than main memory access.***

(f) **I/O Bus**: The I/O bus facilitates communication between the **CPU and the input/output** devices like *keyboards, mice, printers,* etc. Direct Memory Access (DMA): I/O buses sometimes use a feature called DMA, where devices can communicate directly with memory without going through the CPU, reducing CPU workload and improving efficiency.

(g) **Internal Bus**: The internal bus, also known as the local bus, connects internal components within the CPU (such as ALU, registers, cache, and internal memory). It operates at very high speeds and provides rapid data transfer within the CPU.

5. **Instruction Set Architecture (ISA)** The ISA is a set of instructions that the microprocessor can execute. It defines the machine language, including operations like data transfer, arithmetic, logic, control, and I/O. ISAs vary widely, with some offering complex instructions (CISC) and others focusing on a smaller, optimized set (RISC).

6. **Clock and Timing Control** The clock generates pulses that synchronize all operations within the microprocessor. Each pulse initiates a new operation, creating a sequence of actions that execute instructions. Timing control ensures that operations occur at the right moment in coordination with other components, critical for pipelined architectures.

7. **Cache Memory** Many modern microprocessors include one or more levels of cache (L1, L2, L3), which are small, fast memory storage areas close to the CPU cores. Cache stores frequently accessed data to reduce latency, improving overall processing speed by minimizing access to slower main memory. L1, L2, and L3 caches are hierarchical layers of cache memory within a processor, each serving to reduce latency in data access for the CPU and optimize overall system performance. Below are the descriptive details:

   L1: **L1 Cache (Level 1 Cache)** is the fastest and closest cache to the CPU core, designed to store the most frequently accessed data and instructions. Typically the smallest in size, ranging **from 16 KB to 128 KB per core**. Very high speed, with low latency since it's **integrated directly into the CPU core**. It often split into two parts—**one for instructions (L1i)** and another for **data (L1d)**. This allowing simultaneous data and instruction fetches.
   It is used for immediate, very frequent operations, allowing the CPU to quickly access critical instructions and data.

   L2: L2 Cache (Level 2 Cache) serves as an intermediary between the **L1 cache** and the larger, slower **L3 cache** or **main memory**, holding data that is accessed less frequently than L1 but still needed frequently. It is larger than L1, typically ranging from **256 KB to a few MB per core**. It is slower than L1 but faster than L3, providing a balance between speed and storage capacity. It can be integrated within each CPU core or shared among multiple cores, depending on the processor design.
   It holds a larger dataset than L1, storing frequently used data that may not fit in the L1 cache but still needs quick access.

   L3: **L3 Cache (Level 3 Cache)** the largest and slowest cache, typically shared across all cores in a multi-core CPU. It reduces the bottleneck between the CPU and main memory (RAM) by storing data that may be accessed by multiple cores. It is larger than L1 and L2, often ranging from several **MB to tens of**

MB. It is slower than both L1 and L2, but still much faster than main memory (RAM).
It is usually shared across all cores, allowing data sharing and communication between cores, especially in multi-core processors.
It stores less frequently accessed data that doesn't need to be as fast but benefits from faster access than RAM. It's ideal for inter-core data sharing and large datasets that benefit multiple cores.

8. **Pipeline Architecture (in Advanced Microprocessors)** In pipelined microprocessors, instructions are divided into smaller tasks (fetch, decode, execute, etc.), allowing multiple instructions to be processed simultaneously in a staggered manner. This parallelism increases throughput, as the CPU can begin executing the next instruction before completing the previous one.

9. Input/Output (I/O) Interface The I/O interface allows the microprocessor to communicate with external devices, like storage or display devices. It uses specific I/O control signals to manage these communications, which may be handled through dedicated I/O ports or memory-mapped I/O.

10. Memory Management Unit (MMU) The MMU handles the organization and access of memory, particularly in complex or multi-tasking systems. It maps logical addresses (used by programs) to physical addresses (actual locations in memory), enabling virtual memory and protecting processes from interfering with each other.

- **Arithmetic Logic Unit (ALU)**: It handles the arithmetic and logical operations like `addition, subtraction, AND, OR,` etc.

- **Control Unit (CU)**: It manages instruction decoding, execution, and controls data flow within the processor.

- **Registers**: It is small, fast storage locations **inside the CPU** used to hold data temporarily for operations.

- **Cache Memory**: A small, high-speed memory close to the processor, storing frequently used data to improve performance.

- **Buses**: Communication pathways (data, address, and control buses) that connect the CPU to memory and peripherals.

- **Clock Generator**: Provides timing signals that synchronize the microprocessor's operations.

Feature	Cache	Registers
Purpose	Temporary storage for frequently accessed data and instructions, acting as an intermediary between the CPU and main memory	Stores data, instructions, or addresses immediately needed for CPU operations
Location	Located near or within the CPU core, typically in multi-level hierarchy (L1, L2, L3)	Located directly inside the CPU core
Speed	Very high speed but slightly slower than registers	Fastest form of storage in the computer
Size	Small, ranging from a few KB to several MB depending on the level (L1, L2, L3)	Very small, typically a few bytes per register (e.g., 32-bit or 64-bit width)
Access Time	Faster than main memory but slower than registers due to slight latency from being farther from the CPU's ALU (Arithmetic Logic Unit)	Immediate access by the CPU's ALU, with no latency
Cost	Expensive but less costly per byte compared to registers	Most expensive per byte, due to size and placement within the CPU
Management	Managed automatically by the CPU, storing frequently accessed data based on locality of reference	Managed directly by the CPU for specific instructions, with values explicitly loaded and modified by operations
Volatility	Volatile, loses data when power is off	Volatile, loses data when power is off

Table 3: Comparison between Cache and Registers

Feature	Cache Memory	Main Memory (RAM)
Purpose	Temporary storage for frequently accessed data	Primary storage for active processes
Location	Integrated in or near the CPU	Separate from CPU, connected via memory bus
Speed	Very high	Moderate
Size	Small (KB to MB)	Large (GB to TB)
Cost	Higher per byte due to SRAM technology	Lower per byte due to DRAM technology
Technology	SRAM (Static RAM)	DRAM (Dynamic RAM)
Management	Managed automatically by the CPU	Managed by the operating system
Data Retention	Volatile, loses data when power is off	Volatile, loses data when power is off

Table 4: Comparison between Cache Memory and Main Memory (RAM)

3.2 Microprocessor Operation Cycle

The microprocessor operation cycle, also known as the instruction cycle, is the fundamental process by which a microprocessor executes instructions. This cycle is repeated continuously while the computer is running, and it involves three primary stages: Fetch, Decode, and Execute. Each stage plays a critical role in enabling the microprocessor to read, interpret, and perform operations as per the instructions given in the program. The operation in each cycle is discussed below:

1. **Fetch:** This is the initial phase of the operation cycle. The microprocessor retrieves, or fetches, the next instruction from memory. The **Program Counter (PC)** holds the address of the next instruction that needs to be executed. The microprocessor places this address on the address bus, accesses the memory, and reads the instruction at that address.

   The **fetched instruction** is then loaded into the Instruction **Register (IR)**, and the Program Counter is incremented to point to the next instruction. Below is the breakdown of the f etch processes:

   - **Program Counter (PC)**: is a special-purpose register that holds the memory address of the next instruction to be fetched.
   - **Memory Access:** the CPU sends the memory address from the PC to the memory unit. The memory unit retrieves the instruction located at that address.
   - **Instruction Register (IR)**: the fetched instruction is stored in the IR. The IR is a register within the CPU that holds the binary representation of the instruction.
   - **Program Counter Update**: the PC is updated to point to the next instruction in memory. It's typically incremented by the size of the current instruction or by a fixed amount.

2. **Decode:** Once the instruction is fetched or loaded, the microprocessor proceeds to the decode phase. The instruction in the **IR** is decoded (interpreted) to determine the **specific operation** it needs to perform, such as ***arithmetic operations, data movement, logic operations, or control operations***.

   The microprocessor's control unit interprets the instruction, identifies the required operands, and prepares the appropriate circuits within the processor for execution. Below are the breakdown of the decode processes:

   - **Opcode Extraction**: the opcode is extracted from the instruction stored in the IR. The opcode represents the operation to be performed by the CPU.

- **Operand Execution**: if the instruction includes operands, immediate values, or memory addresses, these are extracted during the decode phase. Modes of addressing (e.g., direct, indirect, register) are determined.
- **Control Signal Generation**: The control unit generates control signals based on the opcode and other decoded information. Control signals guide the operation of various components in the CPU, such as the ALU, registers, and buses.

3. **Execute:** This is the final stage, the microprocessor performs the operation specified by the decoded instruction. During execution, the necessary data is accessed, which might involve reading from memory, registers, or input/output devices. The result of the execution may be stored back in memory, in registers, or in an output device.

   Based on the instruction, the Program Counter may also be updated if branching or jumping is required.

   - **Branching (Conditional Jump)**: In ***conditional branching***, the microprocessor evaluates a specific condition, often the result of a prior computation. If the condition is **true**, the microprocessor branches to a different address in memory where the next instruction is located. If the condition is **false**, it proceeds to the next instruction in sequence. This is commonly used in **if-else statements** or **loops** in high-level programming.

     Examples of conditional jump instructions include *JZ (Jump if Zero), JNZ (Jump if Not Zero), JE (Jump if Equal), and JNE (Jump if Not Equal).*

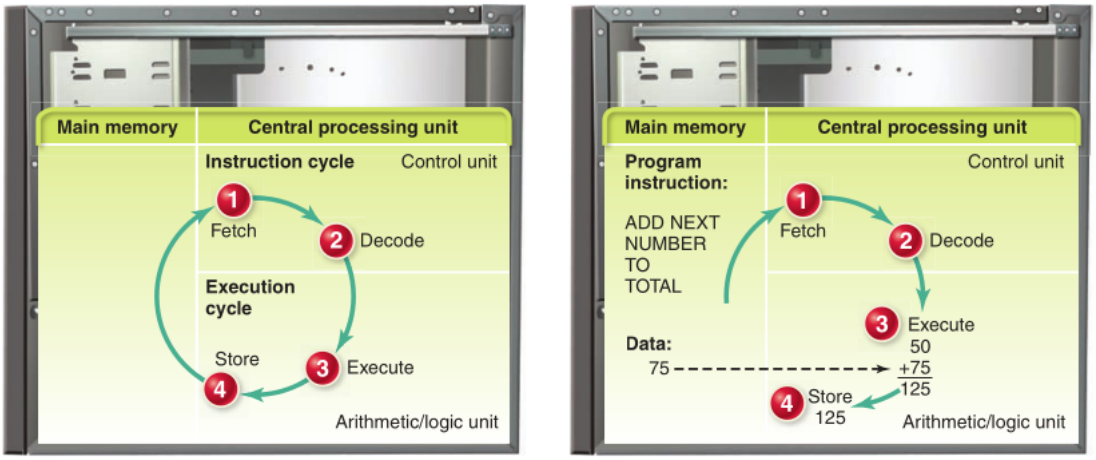
   - **Jumping (Unconditional Jump)**: In an unconditional jump, the microprocessor jumps to a specified address without evaluating any conditions. This is a direct instruction to continue execution from a specific memory address, regardless of any current conditions.

     Unconditional jumps are used for function calls, loops, or simply to transfer control to another part of the program. An example of an unconditional jump instruction is JMP.

4. **Store (Result Storage)**:The results of the operation are stored in registers or memory , depending on the nature of the instruction. Registers may include general-purpose registers, status registers, or special-purpose registers.[10pt] Some computers carry out these four operations one instruction at a time, waiting un til one instruction is processed before another is started. However, most modern per sonal computers are faster than this because they follow a concept called pipelining. ***In pipelining, the CPU does not wait for one instruction to complete the***

ma- chine cycle before fetching the next one. Most PC processors can pipeline up to four instructions.

The fetch-decode-execute cycle can be pipelined to improve performance. ***Pipelining is a technique that overlaps the execution of multiple instructions in order to increase throughput***. In a pipelined microprocessor, the ***fetch, decode, and execute*** stages are performed simultaneously for different instructions. This allows the micropro- cessor to execute more instructions per clock cycle. 11Therefore, The fetch-decode-execute cycle is often divided into pipeline stages in modern microprocessors. Each stage specializes in a specific part of the process, enabling the CPU to process multiple instructions simultaneously. The below figure demonstrates the microprocessor:



Effect of Branching & Jumping in CPU cycle

- In the fetch stage, the microprocessor reads the jump or branch instruction.
- In the decode stage, it determines whether the instruction is conditional or unconditional and if any conditions need to be evaluated.
- During the execute stage, if branching or jumping is required, the Program Counter (PC) is updated to point to the target address rather than the next sequential address.
- This redirection allows the processor to execute code from the new address, effectively altering the program's flow.

Use Case Example

- **Loops**: The microprocessor repeatedly executes a set of instructions by jumping back to the start of the loop until a condition is met.

- **Conditional Execution:** The microprocessor skips or redirects execution based on conditional checks, such as whether a variable is equal to a certain value.
- **Function Calls and Returns:** Jumping allows the microprocessor to execute a separate set of instructions (a function) and then return to the point in the main program after the function completes.

NOTE: *Branching and jumping are fundamental to making microprocessors capable of complex decision-making and control, making programs much more powerful and flexible.*

3.3 Microprocessor intel 8085 vs 8086

The Intel **8085** is an 8-bit microprocessor introduced by Intel around 1976. It's an enhancement of the earlier 8080 processor and is widely used in embedded systems and simple control applications due to its efficiency, simplicity, and low power consumption. Its limited memory addressing and basic instruction set suit it well for tasks that do not require high computational complexity.

The Intel **8086** processor, introduced in 1978, was one of Intel's earliest 16-bit microprocessors, marking a significant development in computing technology. The 8086 was based on a 16-bit architecture, capable of handling 16-bit data in a single cycle. Below are the distinction between Intel 8085 and 8086 processor.

Table 5: Comparison between Intel 8085 and Intel 8086 Microprocessors

Feature	**Intel 8085**	**Intel 8086**
Data Bus Width	8-bit	16-bit
Address Bus Width	16-bit (64KB addressable memory)	20-bit (1MB addressable memory)
Word Size	8-bit	16-bit
Clock Speed	3-5 MHz	5-10 MHz
Registers	8-bit general-purpose registers (B, C, D, E, H, L), 8-bit accumulator, and flags register	16-bit general-purpose registers (AX, BX, CX, DX), 16-bit segment registers (CS, DS, ES, SS), and flags register
Instruction Set	Basic arithmetic, logic, and I/O operations	Advanced instructions with additional arithmetic, logic, and string manipulation operations
Memory Addressing	Direct addressing up to 64KB	Segmented memory addressing up to 1MB
I/O Ports	Supports 256 I/O ports	Supports more complex I/O operations
Interrupts	5 hardware interrupts	256 software interrupts
Applications	Embedded systems, simple control applications	Personal computers, gaming, business computing
Pipeline	No pipeline	Has a 2-stage pipeline (fetch and execute)
Instruction Queue	No instruction queue	6-byte prefetch queue for instruction pipelining
Power Consumption	Lower power consumption	Higher power consumption

3.4 Instruction Pipelining and Superscalar Architecture

They are two techniques that improve the performance of processors by enhancing the way they execute instructions.

3.4.1 Instruction Pipelining

It is a technique used to speed up the execution of instructions by overlapping their processing. In a typical instruction cycle, there are several stages, including *fetching,*

decoding, executing, and writing back (store). Rather than completing each stage for one instruction before starting the next, ***pipelining divides these stages into separate phases, allowing multiple instructions to be processed at once***. Pipelining is commonly found in modern CPUs, where multiple stages (often 5-20) overlap, allowing faster instruction processing.

3.4.2 Superscaler Architechture

It takes pipelining a step further by allowing multiple instructions to be issued and executed in parallel within the same clock cycle. While pipelining enables overlapping execution stages, superscalar architecture enables multiple pipelines or execution units to process multiple instructions at the same time.

Feature	**Instruction Pipelining**	**Superscalar Architecture**
Definition	Divides instruction execution into stages, allowing overlap between stages of different instructions.	Allows multiple instructions to be executed in parallel within the same clock cycle using multiple execution units.
Execution Units	Uses a single pipeline with shared execution units.	Uses multiple pipelines or execution units to handle multiple instructions simultaneously.
Parallelism Type	Provides *intra-instruction parallelism* by overlapping stages of a single instruction across cycles.	Provides *inter-instruction parallelism* by executing multiple instructions in parallel.
Throughput	Increases throughput by reducing the time per instruction.	Further increases throughput by executing more instructions per clock cycle.
Instruction Issue Rate	Typically issues one instruction per clock cycle.	Can issue multiple instructions per clock cycle (e.g., dual-issue, quad-issue).
Dependency Handling	Faces hazards like data, control, and structural hazards due to stage overlap.	Manages additional hazards with techniques like out-of-order execution and register renaming.
Complexity	Simpler in design, generally with a fixed pipeline depth.	More complex, requiring instruction dispatch and reorder buffers for managing multiple pipelines.
Usage	Used in most modern processors to improve instruction throughput.	Common in high-performance processors, increasing instruction throughput by leveraging multiple pipelines.

Table 6: Comparison of Instruction Pipelining vs. Superscalar Architecture

Pipelining focuses on breaking down instruction processing into stages, while **Superscalar** architecture enhances performance by executing multiple instructions in parallel using multiple pipelines. Together, these approaches significantly improve CPU performance and are foundational to modern processor design.

4 Programming Model in Real Mode

The programming model of the Intel 8086 microprocessor in real mode represents the foundational operational framework ***used to execute instructions and manage memory***.

Real mode is the basic operational mode of the 8086 microprocessor. It assumes that the system runs in a *simple environment with direct access to hardware and memory without advanced protection mechanisms*. It was the default mode in the original Intel 8086 and is still used in certain legacy and embedded systems. Below are the basic characteristics:

- Direct access to 1 MB of memory.
- Simple memory segmentation without protection.
- Single-tasking environment.

The programming model of Intel 8086 in real mode is a cornerstone for understanding microprocessor operations, memory addressing, and low-level software design. Its simplicity makes it ideal for embedded systems and learning environments, while its legacy applications continue to influence modern computing architectures. By analyzing real-world examples, such as the BIOS boot process, the enduring relevance of the 8086 real mode programming model becomes evident.

4.1 Registers: General Purpose & Special Purpose

The Intel 8086 microprocessor is a 16-bit processor with a rich set of registers that can be broadly categorized into general-purpose registers, special-purpose registers, and segment registers. These registers facilitate arithmetic, data manipulation, memory addressing, and program control.

4.1.1 General Purpose Register

These registers are versatile and can be used for various purposes, including arithmetic, logical operations, storing data, addresses, and intermediate results. They can be accessed as either 16-bit or 8-bit registers. Each is 16-bit wide but can be accessed as two 8-bit registers (high (**H**)and low (**L**) bytes). Below are the general purpose register used in intel 8086.

- **AX (Accumulator Register)**: It is used for **arithmetic, logic, and data transfer** operations. The most frequently used register in operations. It can be divided into:
  - **AH** (High byte): Upper 8 bits of AX.
  - **AL** (Low byte): Lower 8 bits of AX
- **BX (Base)**: It is used as a base register for addressing memory, as well as for general-purpose data storage. It can be divided into two
  - **BH** (High byte): Upper 8 bits of BX.
  - **BL** (Low byte): Lower 8 bits of BX.
- **CX (Count)**: It is used as a counter for loop control and string operations. It is divided into two
  - **CH** (High byte): Upper 8 bits of CX.
  - **CL** (Low byte): Lower 8 bits of CX
- **DX (Data)**: It is used for input/output operations and as a secondary accumulator for arithmetic (multiplication and division) operations. It is divided into two.
  - **DH** (High byte): Upper 8 bits of DX.
  - **DL** (Low byte): Lower 8 bits of DX

4.1.2 Special Purpose Register

These are tailored for specific functionalities such as program control, flag status, and stack management.

1. **Segment Register**:
   - **CS (Code Segment)**: Holds the segment address of the code.
   - **DS (Data Segment)**: Holds the segment address of the data.
   - **SS (Stack Segment)**: Points to the segment containing the stack.
   - **ES (Extra Segment)**: Additional segment for data operations, especially in string instructions.
2. **Instruction Pointer**: Points to the *address of the next instruction* to be executed.
3. **Flag Register**:
   - Reflects the current state of the processor and the results of operations.

- That is, stores the status flags that indicate the result of arithmetic and logical operations. This includes
  - **Status Flags**: Indicate the result of operations (e.g., Zero Flag, Sign Flag, Carry Flag).
  - **Control Flags**: Control operations (e.g., Direction Flag, Interrupt Flag).

4. **Pointer and Index Registers**:
   - **SP (Stack Pointer)**: Points to the top of the stack in the stack segment.
   - **BP (Base Pointer)**: Used to access stack variables by referencing a base address.
   - **SI (Source Index)**: Points to source data in string operations.
   - **DI (Destination Index)**: Points to destination data in string operations.

Feature	**General-Purpose Registers**	**Special-Purpose Registers**
Primary Use	Multipurpose (arithmetic, logic, data)	Specific tasks (program control, flags, etc.)
Examples	AX, BX, CX, DX	CS, DS, SS, IP, SP, Flags
Bit Addressing	Can be accessed as 16-bit or 8-bit	Accessed as 16-bit only
Flexibility	Highly flexible, can be used in varied operations	Fixed roles tailored for specific functionalities
Memory Addressing	Rarely used directly for addressing	Includes registers dedicated to addressing (CS, DS, etc.)
Role in Instructions	Commonly used in arithmetic and loops	Key for program execution flow and control

Table 7: Comparison of General-Purpose and Special-Purpose Registers in Intel 8086

NOTE: ***The general-purpose registers in the Intel 8086 provide flexibility for computations and temporary data storage, while the special-purpose registers focus on managing program control, memory segmentation, and processor state.***

4.1.3 Memory and Addressing Modes

The Intel 8086 memory has a 20-bit address bus, allowing it to address up to $2^{20} = 1MB$ of memory. However, its 16-bit architecture imposes constraints on the immediate

addressing capabilities, which is resolved using segmentation.

Segmentation: Memory is divided into segments, each of size 64KB. The segmentation model allows the CPU to address 1MB memory with only 16-bit registers. The Segment Registers consist of the following as discussed in the above session:

- **CS (Code Segment)**: Stores the base address of the code segment.
- **DS (Data Segment)**: Stores the base address of the data segment.
- **SS (Stack Segment)**: Stores the base address of the stack segment.
- **ES (Extra Segment)**: Provides an additional data segment for operations like string handling.

Effective Address Calculation: A physical address is calculated by combining the segment base address (from segment registers) and an offset address (from general-purpose or pointer registers): It is calculated as follows:

$$PhysicalAddress = (SegmentReigster * 16) + offset$$

The physical address calculation involves combining the contents of a segment register (CS, DS, ES, or SS) with an offset value to form a 20-bit physical address. With this, we can effectively work with memory addressing in 8086 assembly language programming.
NOTE:

- `The offset is an additional value added to the segment base address to calculate the physical address of the exact memory location being accessed.`
- `The offset is always relative to the segment's base address.`
- `Its maximum value is FFFFH (16-bit), which allows addressing the full 64 KB within a segment.`
- `By combining the segment and offset, the 8086 achieves its 20-bit physical addressing, covering a total of 1 MB memory space.`

4.1.4 Question 1

An 8086 processor uses segmented memory addressing, where the physical address is calculated using a segment register (e.g., CS, DS, ES, or SS) and an offset. The Segment register is `0x1234`, and offset is `0x5678`. Calculate the physical address.

Solution

- Convert the segment and offset into decimal (necessary)
  - $Segment = 0x1234(hexadecimal) = 4660(decimal)$.
  - $Offset = 0x5678(hexadecimal) = 22136(decimal)$.
- Multiply the segment value by 16 (or shift left by 4 bits):
  - $Segment\ Base\ Address = 466016 = 74560(decimal) = 0x12340(hexadecimal)$
- Add the offset to the segment base address:
  - $Physical\ Address = 0x12340 + 0x5678 = 0x179B8(hexadecimal)$

```
1    MOV AX, 0x1234     ; Load segment value into AX
2    MOV DS, AX         ; Move segment value into the Data
       Segment register
3    MOV SI, 0x5678     ; Load offset value into Source Index
       register
4    MOV AL, [SI]       ; Access the byte at physical address 0
       x179B8
```

NOTE:

- In the 8086, each segment starts at an address that is a multiple of 16 (or 0x10 in hexadecimal). Thus, the segment value is shifted left by 4 bits to form the segment base address.
- Since the segment is 16 bits and the offset is also 16 bits, the maximum physical address is:

$$Max\ Physical\ Address = 0xFFFF0 + 0xFFFF = 0x10FFEF$$

- This gives a total addressable memory of 1 MB.

Question 1: Effective Address with Displacement Only

Given the instruction:

`MOV AL, [1234H]`

Assume the `DS` register contains `1000H`. Calculate the physical address accessed by the instruction.

Solution: The effective address is given as `1234H`. The physical address is calculated as:

$$\text{Physical Address} = \texttt{DS} \times 16 + \texttt{EA}$$

Substitute the values:

$$\text{Physical Address} = 1000H \times 16 + 1234H = 10000H + 1234H = 11234H$$

Answer: The physical address accessed is `11234H`.

Question 2: Effective Address with Base Register

Given the instruction:

```
MOV BX, [BP]
```

Assume `SS = 2000H` and `BP = 0020H`. Calculate the physical address accessed.

Solution: The effective address is determined by the value in the `BP` register:

$$\texttt{EA} = \texttt{BP} = 0020H$$

The segment register used with `BP` is `SS`. The physical address is:

$$\text{Physical Address} = \texttt{SS} \times 16 + \texttt{EA}$$

Substitute the values:

$$\text{Physical Address} = 2000H \times 16 + 0020H = 20000H + 0020H = 20020H$$

Answer: The physical address accessed is `20020H`.

Question 3: Effective Address with Base and Index Registers

Given the instruction:

```
MOV AL, [BX+SI]
```

Assume `DS = 3000H`, `BX = 0100H`, and `SI = 0020H`. Calculate the physical address accessed.

Solution: The effective address is computed by adding the contents of `BX` and `SI`:

$$\texttt{EA} = \texttt{BX} + \texttt{SI} = 0100H + 0020H = 0120H$$

The physical address is:

$$\text{Physical Address} = \texttt{DS} \times 16 + \texttt{EA}$$

Substitute the values:

$$\text{Physical Address} = 3000H \times 16 + 0120H = 30000H + 0120H = 30120H$$

Answer: The physical address accessed is `30120H`.

Question 4: Effective Address with Base, Index, and Displacement

Given the instruction:

```
MOV AX, [BX+DI+0050H]
```

Assume `DS = 4000H`, `BX = 0100H`, and `DI = 0030H`. Calculate the physical address accessed.

Solution: The effective address is computed as:

$$\texttt{EA} = \texttt{BX} + \texttt{DI} + \texttt{Displacement}$$

Substitute the values:

$$\texttt{EA} = 0100H + 0030H + 0050H = 0180H$$

The physical address is:

$$\text{Physical Address} = \texttt{DS} \times 16 + \texttt{EA}$$

Substitute the values:

$$\text{Physical Address} = 4000H \times 16 + 0180H = 40000H + 0180H = 40180H$$

Answer: The physical address accessed is `40180H`. **Addressing modes** determine how the microprocessor accesses data stored in memory, registers, or provided as immediate values. Common addressing modes are:

1. **Immediate Addressing:** The operand is directly included in the instruction itself. It is used where we have constant or fixed value

```
MOV AX, 1234h  ; Load the immediate value 1234h into AX register
```

2. **Register Addressing Mode**: The operand is stored in a register, and the instruction specifies the register. It is used for fast data manipulation within registers.

```
MOV BX, AX  ; Copy the contents of AX into BX
```

3. **Direct Addressing Mode**: The effective address of the operand is specified explicitly in the instruction. It is used access specific memory locations.

```
MOV AX, [1234h]  ; Load the value from memory
   address 1234h into AX
```

4. **Register Indirect Addressing Mode:** The effective address is stored in a base or index register. It is used to access memory indirectly via registers.

```
MOV AX, [BX]  ; Load the value from the memory address in
    BX into AX
```

4.2 Real Mode and Protected Mode

Real Mode: Intel 8086 processors operate in Real Mode, but later processors in the x86 family introduced Protected Mode, starting with the Intel 80286. These two modes are fundamental in understanding how x86 processors manage memory, access instructions, and interact with the system.

Protected Mode: It was introduced with the Intel 80286 processor and enhanced in later processors. It overcomes many limitations of Real Mode, providing advanced features for modern operating systems.

Feature	**Real Mode**	**Protected Mode**
Memory Addressing	20-bit address bus, can address up to 1 MB of memory.	32-bit address bus, can address up to 4 GB of memory.
Segmentation	Fixed 64 KB segments, addressing is limited to segment:offset.	Allows segmentation and paging for efficient memory use.
Protection	No memory protection; all programs have unrestricted access to memory.	Provides hardware-level memory protection and privilege levels.
Multitasking	Not supported; only one task runs at a time.	Supports multitasking through task switching.
Mode Activation	Default mode after CPU reset; no setup is required.	Requires enabling in the CR0 register (Control Register).
Interrupt Handling	Uses the interrupt vector table located in the first 1 KB of memory.	Advanced interrupt descriptor table with more flexibility.
Virtual Memory	Not supported.	Supported via paging mechanisms.
Processor Access	Direct hardware access; no privilege levels.	Implements privilege levels (rings 0–3) for better security.

Feature	Real Mode	Protected Mode
Backward Compatibility	Fully compatible with older software written for 8086.	Not backward-compatible with 8086 real mode directly.
Use Case	Simple operating systems, embedded systems, and initial boot loaders.	Modern operating systems and complex applications.

Real Mode is simplistic and retains historical significance for its role in early computing. However, it lacks the sophistication required for modern computing environments. **Protected Mode** introduces advanced features such as memory protection, multitasking, and hardware-level privilege enforcement, making it the standard for contemporary operating systems and applications.

5 Interrupt Handling and Prioritization

Interrupts are mechanisms that allow devices or software to temporarily ***halt the normal flow of a program to attend to a high-priority event***. Interrupt handling and prioritization are critical components in real-time and embedded systems to ensure timely. Interrupt handling and prioritization are indispensable (needed) for building robust and responsive systems, ensuring critical tasks are handled efficiently while balancing system resources.

The Intel 8086 supports 256 interrupts, identified by an interrupt vector number (0–255). These are stored in the Interrupt Vector Table (IVT), located at the start of memory (0000H to 03FFH). Each entry in the IVT contains a 4-byte pointer (segment:offset) to the interrupt handler.

5.1 Interrupt Handling Process

Interrupts are a mechanism used by the 8086 processor to handle asynchronous events that require immediate attention. The processor's interrupt handling system enables it to pause the execution of the current program, execute a predefined interrupt service routine (ISR), and then resume the original program execution.

The interrupt handling process in the 8086 processor involves several steps which are:

1. Interrupt Recognition

When an interrupt occurs, the processor completes the current instruction before recognizing the interrupt. The recognition process depends on the type of interrupt:

- **Non-Maskable Interrupt (NMI):** Recognized immediately since it cannot be disabled.

- **Maskable Interrupt (INTR):** Checked at the end of every instruction cycle, provided the interrupt flag (`IF`) is set.
- **Software Interrupts:** Triggered by the execution of an `INT` instruction in the program.

2. Saving the Current State

Before handling the interrupt, the processor saves the state of the current program. The following steps are performed:

1. The contents of the `FLAGS` register are pushed onto the stack.
2. The code segment (`CS`) register is pushed onto the stack.
3. The instruction pointer (`IP`) register is pushed onto the stack.

3. Determining the Interrupt Vector

The 8086 uses an **Interrupt Vector Table (IVT)** to locate the ISR for a specific interrupt. The IVT is a 1KB table stored at memory addresses `0000H` to `03FFH`. Each interrupt type corresponds to a 4-byte entry in the IVT:

- The first 2 bytes store the offset address of the ISR.
- The next 2 bytes store the segment address of the ISR.

The processor multiplies the interrupt type number by 4 to find the appropriate entry in the IVT.

4. Jumping to the ISR

Using the address fetched from the IVT, the processor loads the `CS` and `IP` registers with the segment and offset of the ISR. It then jumps to the ISR and begins executing the interrupt service routine.

5. Executing the ISR

The ISR performs the required operations to handle the interrupt. For example:

- Handling keyboard input.
- Responding to a timer event.
- Completing a device I/O operation.

The ISR must end with the `IRET` instruction to return control to the interrupted program.

6. Restoring the Previous State

When the `IRET` instruction is executed, the processor restores the previously saved state:

1. The `IP` is popped from the stack.
2. The `CS` is popped from the stack.
3. The `FLAGS` are popped from the stack.

After restoring the state, the processor resumes execution of the interrupted program.

5.2 Flowchart of the Interrupt Handling Process

1. Detect an interrupt request.
2. Complete the current instruction.
3. Push `FLAGS`, `CS`, and `IP` onto the stack.
4. Fetch the interrupt vector address.
5. Load the `CS` and `IP` registers with the ISR address.
6. Execute the ISR.
7. Execute `IRET` to return to the original program.

Classification of Interrupts

Interrupts in the Intel 8086 can be categorized based on their source and characteristics as follows:

1. Hardware Interrupts

Hardware interrupts are generated by external devices to gain the processor's attention. These can be further divided into:

- **Maskable Interrupt (INTR):** Can be enabled or disabled using the `IF` (Interrupt Flag). Example: I/O devices.
- **Non-Maskable Interrupt (NMI):** Cannot be disabled and is used for high-priority events like power failure or memory errors.

2. Software Interrupts

Software interrupts are triggered by the execution of the `INT` instruction in a program. These are used to invoke predefined interrupt service routines (ISRs) or custom routines. For example:

- `INT 21H`: A DOS interrupt for various system calls.

3. Exceptions

Exceptions are internally generated by the processor during the execution of instructions. Examples include:

- Divide by zero error.
- Invalid opcode execution.

5.3 Interrupt Types

The following table provides a detailed overview of different types of interrupts in the 8086 with examples:

Interrupt Type	Interrupt Source	Example	Description
Hardware Interrupt (Maskable)	External device	Keyboard interrupt	Generated when a key is pressed. Requires `IF=1` to be recognized.
Hardware Interrupt (Non-Maskable)	External device	Power failure detection	Triggered by critical events that require immediate action.
Software Interrupt	`INT` instruction	`INT 21H`	Used for DOS system calls like file handling and console I/O.
Exception (Divide Error)	Arithmetic operation	Division by zero	Triggered when the divisor is zero in a division instruction.
Exception (Invalid Opcode)	Faulty instruction	Invalid opcode	Occurs when an unrecognized opcode is encountered during execution.

Table 9: Types of Interrupts in Intel 8086 with Examples

Example 1: Maskable Interrupt (`INTR`)

A maskable interrupt can be generated by a timer to signal the completion of an interval. The ISR for this interrupt can update a clock display.

Example 2: Software Interrupt (`INT 21H`)

The `INT 21H` interrupt is used in DOS for various system services. For example, to display a string:

```
MOV AH, 09H ; Function to display string
LEA DX, MSG ; Load address of message
INT 21H     ; Call DOS interrupt
```

Example 3: Divide Error Exception

An example of a divide error exception is executing the following:

```
MOV AX, 10H
MOV BL, 00H
DIV BL ; Triggers divide error exception
```

Interrupt	**Function**	**Example Code (Assembly)**	**Meaning / Application**

INT 21H	Display a character (AH = 02H)	`MOV AH, 02H`
		`MOV DL, 'A'`
		`INT 21H`
			Outputs a single character (e.g., 'A') to the screen.

INT 21H	Read a character (AH = 01H)	`MOV AH, 01H` `INT 21H`	Reads a single character from the keyboard and stores it in the AL register.

Interrupt	**Function**	**Example Code (Assembly)**	**Meaning / Application**

INT 21H	Open a file (AH = 3DH)	`MOV AH, 3DH`
		`MOV DX, OFFSET FILENAME`
		`MOV AL, 00H ; Read-only`
		`INT 21H`
			Opens a file and returns the file handle in AX.

INT 21H	Write to a file (AH = 40H)	`MOV AH, 40H`
		`MOV BX, HANDLE`
		`MOV CX, LENGTH`
		`MOV DX, OFFSET BUFFER`
		`INT 21H`
			Writes data to a file using a handle, buffer, and length.

INT 21H	Get system date (AH = 2AH)	`MOV AH, 2AH` `INT 21H`	Retrieves the system date and stores it in CX (year), DH (month), and DL (day).

INT 21H	Terminate program (AH = 4CH)	`MOV AH, 4CH`
		`MOV AL, 00H ; Exit code`
		`INT 21H`
			Terminates the program and returns the exit code to the operating system.

INT 10H	Set video mode (AH = 00H)	`MOV AH, 00H`
		`MOV AL, 03H ; Text mode`
		`INT 10H`
			Changes the video mode, e.g., setting the screen to text mode (03H).

Interrupt	Function	Example Code (Assembly)	Meaning / Application

INT 13H	Read sector from disk	`MOV AH, 02H`
		`MOV AL, 01H ; Number of sectors`
		`MOV CH, 00H ; Cylinder`
		`MOV CL, 01H ; Sector`
		`MOV DH, 00H ; Head`
		`MOV DL, 80H ; Drive`
		`INT 13H`
			Reads a sector from the specified disk drive and location.

INT 16H	Keyboard input (AH = 00H)	`MOV AH, 00H` `INT 16H`	Waits for a keypress and stores the ASCII code in AL and scan code in AH.

Table 10: DOS interrupt with example

5.4 Organisation of the Interrupt System: Vectors, External Interrupts

As we all know that ***interrupt system in a microprocessor allows the processor to respond to urgent events or requests, halting its current operation to execute a corresponding service routine and then resuming the main task***. The organization of this system is critical to ensure efficient handling of interrupts.

The interrupt system in the Intel 8086, with its IVT and support for external interrupts via the INTR and NMI pins, allows for efficient and flexible handling of events. The use of an interrupt controller like the 8259A enhances this system by providing prioritization and expandability. This structured approach ensures that interrupts are handled seamlessly, making the processor capable of real-time operations.

5.5 Interrupt Vectors

The **Interrupt Vector Table (IVT)** is a reserved memory region that holds pointers (vectors) to *Interrupt Service Routines (ISRs)*. Each vector corresponds to an interrupt number and contains the address of the ISR that should handle the interrupt.

5.5.1 Structure of the Interrupt Vector Table

- The IVT is located in memory from `0000:0000H` to `0000:03FFH`, occupying 1 KB.
- Each entry in the table is 4 bytes:
  - 2 bytes for the offset address.
  - 2 bytes for the segment address.
- The vector address is calculated as:

$$\text{Vector Address} = \text{Interrupt Number} \times 4$$

- For example, interrupt number `21H` corresponds to the vector at `21H` $\times$ 4 = `0084H`.

5.5.2 Types of Interrupts

- **Hardware Interrupts**: Triggered by external devices (e.g., keyboards, timers).
- **Software Interrupts**: Triggered by the `INT` instruction (e.g., `INT 21H` for DOS calls).

5.5.3 ISR Execution

When an interrupt occurs:

1. The processor fetches the vector from the IVT.
2. The ISR address is loaded into the Instruction Pointer (IP) and Code Segment (CS).
3. The ISR executes and ends with the `IRET` instruction, returning control to the interrupted task.

5.6 External Interrupts

External interrupts are signals generated by hardware devices to request the processor's attention. These interrupts are asynchronous and independent of the current instruction flow.

5.6.1 Interrupt Request (IRQ) Lines

- The 8086 microprocessor has two external interrupt pins:
  - **INTR (Interrupt Request)**: A general-purpose interrupt line.

  - **NMI (Non-Maskable Interrupt)**: A high-priority interrupt line that cannot be disabled.

- External devices connect to these pins to signal interrupt requests.

5.6.2 Interrupt Controller (8259A)

To manage multiple external devices, the **8259A Programmable Interrupt Controller (PIC)** is used.

- The PIC prioritizes interrupts and communicates with the processor via the `INTR` line.
- Features of the 8259A:
  - Supports up to 8 interrupt lines, expandable to 64 through cascading.
  - Handles priority and vector generation.

5.7 NMI vs INTR

The following table compares the characteristics of `NMI` and `INTR`:

Feature	NMI	INTR
Maskability	Non-maskable	Maskable
Priority	Higher priority	Lower priority
Use Case	Critical events (e.g., power failure)	General-purpose interrupts

5.7.1 External Interrupt Handling Steps

1. The external device asserts the interrupt signal (`INTR` or `NMI`).
2. For `INTR`, the processor sends an **Interrupt Acknowledge (INTA)** signal.
3. The PIC provides the interrupt vector number.
4. The ISR address is fetched, and the ISR executes.
5. After execution, control returns to the main program using the `IRET` instruction.

5.8 Priority and Nesting

5.8.1 Priority

Interrupts are prioritized based on their type and source:

- `NMI` has the highest priority.
- Maskable interrupts are prioritized by the PIC.

5.8.2 Nesting

Higher-priority interrupts can interrupt lower-priority ISRs if interrupt nesting is enabled. This is controlled using the **Interrupt Flag (IF)**.

5.9 Applications

- **Real-Time Systems**: Immediate response to critical events.
- **Peripheral Communication**: Handles data from devices like keyboards or printers.
- **Error Handling**: Responds to critical errors, such as memory faults.
- **Efficient CPU Utilization**: Reduces polling overhead.

Note

The interrupt system in the Intel 8086 is a well-organized mechanism that combines interrupt vectors, external interrupts, and controllers like the 8259A to handle diverse tasks efficiently. This system enables the processor to perform real-time operations while maintaining flexibility and responsiveness to external events.

5.10 Pin Configuration of Intel 8086

The Intel 8086 processor has a 40-pin dual in-line package (DIP) configuration. These pins are categorized based on their functionality, including address/data bus, control signals, power, and clock.

Pin Number	Pin Name	Description
1–16	`AD0--AD15`	Address/Data bus: These are multiplexed pins used to carry the address during the first clock cycle and data during subsequent cycles.
17	`NMI`	Non-Maskable Interrupt: A high-priority interrupt that cannot be disabled. It is typically used for critical tasks like power failure detection.
18	`INTR`	Interrupt Request: A general-purpose interrupt pin used by external devices to request service from the processor.

Pin Number	Pin Name	Description
19	CLK	Clock: Provides the clock input to synchronize the processor's operations.
20	GND	Ground: Provides the reference voltage for the processor.
21	READY	Ready: Used by slower memory or I/O devices to pause the processor until they are ready to communicate.
22	RESET	Reset: Initializes the processor and starts program execution from the address FFFF0H.
23	TEST	Test: Used for debugging purposes in conjunction with the WAIT instruction.
24	MN/$\overline{MX}$	Minimum/Maximum Mode: Determines the processor's operating mode. High logic selects minimum mode, while low logic selects maximum mode.
25–30	A16--A19/S3--S6	Address/status lines: Carry higher-order address bits during memory access and status signals during instruction execution.
31	BHE/$\overline{S7}$	Bus High Enable/Status: Indicates the transfer of data on the upper half (D8–D15) of the data bus.
32	RD	Read: Active low signal indicating the processor is reading data from memory or an I/O device.
33	READY	Signals the processor to wait for slower devices.
34	INTA	Interrupt Acknowledge: Acknowledges an interrupt request from an external device.
35	ALE	Address Latch Enable: Enables external latching of the address lines.
36	DT/$\overline{R}$	Data Transmit/Receive: Used in maximum mode to control data flow direction.
37	DEN	Data Enable: Enables the external data bus transceivers in maximum mode.
38	HOLD	Hold: Indicates that another bus master is requesting control of the system bus.

Pin Number	Pin Name	Description
39	HLDA	Hold Acknowledge: Acknowledges the HOLD request and indicates that the bus is relinquished.
40	VCC	Power Supply: Provides the required voltage to the processor (+5V).

5.11 Modes of Operation

The pin configuration allows the Intel 8086 to operate in two modes:

- **Minimum Mode**: For small, single-processor systems. Control signals are generated internally.
- **Maximum Mode**: For multi-processor systems. Control signals are generated externally using a bus controller (e.g., 8288).

5.11.1 Interrupt Service Routine (ISR)

The Interrupt Service Routine (ISR) is a critical concept in the Intel 8086 processor, enabling it to handle interrupts efficiently. ISR plays a critical role in the processor's interrupt handling mechanism.

NOTE: *The ISR is a specific block of code or function designed to handle an interrupt event. Interrupts are signals to the processor that require immediate attention, and the ISR contains the logic to respond to these signals.*

ISR works as follows:

- **Interrupt Request**: When an interrupt is triggered (either hardware or software), the 8086 processor halts the current execution and transfers control to the ISR.
- **Interrupt Vector Table (IVT)**: The processor determines the address of the ISR by consulting the Interrupt Vector Table (IVT), a table located in memory at the base address 0000:0000. Each interrupt type has a specific entry in the IVT that contains the segment and offset of the corresponding ISR.

  The IVT is a fixed memory region in the 8086 processor, located at 0000:0000 to 0000:03FF (1 KB in size). Each entry in the IVT contains a 4-byte pointer (2 bytes for the offset and 2 bytes for the segment) that points to the corresponding ISR.

  There are 256 interrupt types (0-255) in the 8086.

  - Type 0 to 31 are generally reserved for processor-defined or system exceptions (e.g., divide-by-zero).

– Type 32 to 255 can be used for user-defined ISRs.

For instance, a software interrupt INT n directly maps to the nth entry in the IVT.

- **Saving State**: The processor saves the current execution state (IP, CS, and FLAGS) on the stack before jumping to the ISR. This ensures the processor can resume the interrupted program after servicing the interrupt.
- **Executing ISR**: The ISR executes the required operations to handle the interrupt.
- **Return to Main Program**: After servicing the interrupt, the IRET (Interrupt Return) instruction is used in the ISR. This restores the processor's state from the stack and resumes execution of the interrupted program.m

The table below lists common interrupt types in the Intel 8086 processor, their purpose, and usage.

Interrupt Number	**Purpose**	**Description/Usage**
0 (Type 0)	Divide Error	Raised when a division by zero or an overflow occurs in a division operation.
1 (Type 1)	Single-Step Interrupt	Triggered after each instruction when the trap flag is set for debugging.
2 (Type 2)	Non-Maskable Interrupt (NMI)	Used for critical hardware failures like parity errors or power loss.
3 (Type 3)	Breakpoint Interrupt	Used by debugging software to stop program execution at a specific point.
4 (Type 4)	Overflow Interrupt	Triggered by the 'INTO' instruction when the overflow flag is set.
5 (Type 5)	Bounds Check Interrupt	Raised when an array index is out of bounds using the 'BOUND' instruction.
6 (Type 6)	Invalid Opcode	Triggered when an undefined or invalid instruction is executed.
8 (Type 8)	Double Fault	Raised when an exception occurs while handling another exception.
10-11	Reserved	Reserved for future use or system-specific interrupts.
16 (Type 16)	Keyboard Interrupt	Handles keyboard inputs via the interrupt vector.
32-47	Hardware IRQs (Interrupt Requests)	Reserved for hardware-related interrupts like timers and external devices.

Interrupt Number	Purpose	Description/Usage
0x10 (INT 10H)	Video Services	Provides video-related services like mode setting, character display, etc.
0x13 (INT 13H)	Disk Services	Used for low-level disk operations like read, write, and format.
0x16 (INT 16H)	Keyboard Services	Provides services to handle keyboard input and buffer checks.
0x17 (INT 17H)	Printer Services	Manages printer-related tasks such as initialization and data output.
0x19 (INT 19H)	Bootstrap Loader	Used during system startup to load the operating system.
0x1A (INT 1AH)	Real-Time Clock Services	Reads or sets the system's real-time clock.
0x21 (INT 21H)	DOS Services	Provides a wide range of DOS functions like file management, I/O handling, etc.
0x80-0xFF	User-Defined	Reserved for custom use in user-defined ISRs.

5.12 Hardware vs. Software Interrupts

5.12.1 Hardware Interrupts

These are interrupts generated by external devices to signal the processor for attention. They are asynchronous events and can occur at any time during the program execution. Hardware interrupts features are discussed below:

- Triggered by external hardware devices like keyboards, timers, printers, etc.
- Cannot be controlled by software directly.
- Requires interrupt lines (pins) on the processor.

Hardware interrupts is divided into two types **Non-Maskable** and **Maskable** interrupts:

1. **Non-Maskable Interrupt (NMI)**:
    - Connected to the NMI pin of the 8086.
    - High-priority interrupt that cannot be disabled.
    - Used for critical tasks like hardware failure or power failure.
    - Vector address: INT 2 (0008H in the interrupt vector table).
2. **Maskable Interrupt (INTR)**:

- Connected to the INTR pin of the 8086.
- Lower-priority interrupt that can be disabled using the Interrupt Flag (IF) in the flags register.
- Vector address determined by an external Interrupt Controller (like 8259 PIC).

Interrupt Code	Explanation	Example
INT 2 (Non-Maskable Interrupt)	Triggered by the **NMI** pin. This interrupt cannot be masked or disabled and is used for high-priority tasks such as critical hardware failures.	Example: Handling power failure signals in critical systems. ISR is located at address **0008H**.
INTR (Maskable Interrupt)	Triggered via the **INTR** pin and prioritized using an external Programmable Interrupt Controller (PIC), such as the 8259. Maskable via the Interrupt Flag (IF).	Example: Keyboard input handling. The interrupt vector is determined by the PIC and points to the ISR for handling the key press.
Timer Interrupt	Generated by the system timer, typically routed through the PIC. Used for timekeeping and task scheduling.	Example: A system clock that triggers every millisecond to update the time. Interrupt vector is determined by the PIC setup.
Hardware Device-Specific Interrupts	Generated by specific hardware peripherals such as printers, disks, or serial ports, routed through the PIC.	Example: Disk drive requests for data read/write operations. Interrupt vector is dynamically assigned based on the controller configuration.

Table 13: Hardware Interrupts in Intel 8086: Codes, Explanations, and Examples

5.12.2 Software interrupts

These are interrupts generated by the execution of an INT instruction in a program. They are synchronous and under the programmer's control. Below are the features:

- Triggered explicitly by software using the INT instruction.
- Used to access system services like file operations, printing, etc.
- Allows programmers to invoke predefined routines.

Common Software interrupts used in intel 8086 are:

- **INT 21H**: DOS interrupt for system services.
- **INT 10H**: Video services.
- **INT 13H**: Disk services.
- **INT 16H**: Keyboard services.

The table below provides details of commonly used software interrupts in the Intel 8086 processor.

Interrupt Code	Explanation	Example
`INT 10H`	Used for video services such as setting video modes, cursor control, and text manipulation.	Set video mode to 80x25 color text: `MOV AH, 00H`
`MOV AL, 03H` `INT 10H`		
`INT 13H`	Provides low-level disk services such as reading, writing, and formatting.	Read a sector from the disk: `MOV AH, 02H`

`MOV AL, 01H`
`MOV CH, 00H`
`MOV CL, 02H`
`MOV DH, 00H`
`MOV DL, 80H`
`INT 13H`		

`INT 16H`	Used for keyboard services like reading a keystroke or checking the keyboard buffer status.	Read a key from the keyboard: `MOV AH, 00H`
`INT 16H`		
`INT 17H`	Provides printer services such as initializing, checking printer status, and sending data to the printer.	Initialize the printer: `MOV AH, 01H`
`MOV DX, 00H` `INT 17H`		

Interrupt Code	Explanation	Example
`INT 21H` ... `INT 21H`	Offers DOS interrupt services such as file management, device I/O, and program termination.	Terminate a program: `MOV AH, 4CH`
`INT 1AH` ... `INT 1AH`	Provides real-time clock services, such as reading the system clock or setting it.	Read the system clock: `MOV AH, 00H`

5.12.3 Multiple Interrupts Handling in Modern Systems

Handling multiple interrupts in modern systems differs significantly from how it was managed in the Intel 8086 microprocessor due to advancements in hardware and software.

Modern systems have evolved to handle multiple interrupts efficiently using advanced interrupt controllers, dynamic priorities, and support for multicore processing, overcoming the limitations of the Intel 8086. While the 8086 was a foundational architecture for understanding interrupt handling, modern techniques are far more robust and scalable, essential for complex and real-time applications.

APIC (Advanced Programmable Interrupt Controller) and **GIC** (Generic Interrupt Controller) are advanced interrupt controllers designed for modern processors and systems. They significantly enhance interrupt handling compared to traditional methods, especially in multicore and multiprocessor environments. Below is an overview of each:

Feature	**Intel 8086 Microprocessor**	**Modern Systems/Microprocessors**
Interrupt Vectoring	Static IVT, fixed memory locations	Dynamic ISR allocation with flexibility for changes
Interrupt Controllers	Basic interrupt handling logic	Advanced controllers (e.g., APIC , GIC for complex prioritization and distribution
Interrupt Nesting	Requires manual context saving/restoration	Automatic with hardware support for nested interrupts
Priority Handling	Fixed priority based on interrupt type	Dynamic, preemptive priority schemes

Feature	**Intel 8086 Microprocessor**	**Modern Systems/Microprocessors**
Concurrency	Single interrupt handled at a time	Interrupts distributed across multiple cores in multicore systems
Latency	Higher latency due to simpler architecture	Optimized for minimal latency and faster response
Scalability	Limited to a few hardware sources	Supports thousands of interrupt sources with advanced controllers
Real-Time Support	Absent	Widely available in real-time operating systems (RTOS)
Virtualization Support	Not supported	Full support for virtualized interrupts and isolation in virtual machines

Both **APIC** and **GIC** are critical for interrupt handling in modern systems, tailored to their respective processor architectures (x86 and ARM). APIC excels in symmetric multiprocessing (SMP) environments like servers and desktops, while GIC is designed for scalability and low-power requirements in embedded and mobile platforms. These controllers reflect the evolution of interrupt handling, enabling efficient management of complex and high-volume interrupt scenarios in diverse computing environments.

6 Memory Interfacing and Address Decoding

Memory interfacing in the Intel 8086 microprocessor involves connecting external memory to the processor, enabling it to read and write data or instructions. Address decoding ensures that only the intended memory location is accessed during these operations.