İSTÜN

İSTANBUL SAĞLIK VE TEKNOLOJİ
ÜNİVERSİTESİ

ISTANBUL HEALTH AND TECHNOLOGY UNIVERSITY

FACULTY OF ENGINEERING AND NATURAL SCIENCE

COMPUTER ENGINEERING DEPARTMENT

CSE301 COMPUTER ARCHITECTURE COURSE

FALL 2025-2026

ASSIGNMENT 2 : Design and Comparative Analysis of I/O Mechanisms

STUDENT NAME: Ümmügülsün Türkmen

STUDENT NUMBER: 230611056

DUE DATE: 28/12/2025

Table of Contents

Section 1: Introduction

The Architectural Necessity of I/O Modules

In a modern computer system, the Input/Output (I/O) subsystem serves as the essential bridge between the internal high-speed components (the Processor and Main Memory) and the diverse world of external peripheral devices. While the internal system operates on nanosecond scales using synchronized electronic signals, the external environment consists of devices that vary wildly in terms of data rates, data formats, and operational principles.

As discussed in our lectures, a direct connection between a peripheral device and the system bus is architecturally impractical for three fundamental reasons:

1. **Peripheral Variety and Complexity:** There is an immense variety of external devices—ranging from simple human-interface devices like keyboards to high-speed networking cards and complex storage systems. Incorporating the specific control logic for every possible device directly into the CPU would result in an over-complicated and inefficient processor design. The I/O module acts as an **abstraction layer**, shielding the CPU from the messy electrical and mechanical details of each peripheral.
2. **The "Speed Mismatch" Phenomenon:** There is a monumental gap between the internal speed of the computer and the operational speed of the external world. As emphasized in the lecture, a CPU can execute billions of instructions per second, while a human typing on a keyboard or a mechanical disk drive moving its head operates in milliseconds or seconds. Without an I/O module to act as a **data buffer**, the CPU would be "paralyzed", wasting millions of cycles waiting for slow devices to respond.
3. **Data Format and Word Length Conversion:** External devices often use different data formats (e.g., serial vs. parallel) and word lengths than the computer's internal architecture. The I/O module performs the necessary "translation" to ensure that the data arriving at the system bus matches the internal expectations of the architecture.

In this report, we will critically analyze the evolution of I/O techniques—from the basic Programmed I/O to the highly efficient Direct Memory Access (DMA) and modern Direct Cache Access (DCA) technologies. We will evaluate how these mechanisms manage processor overhead, maintain system throughput, and adapt to the demands of high-speed modern networking.

This section provides a deep dive into the fundamental logic of Input/Output architectures, focusing on why specialized modules are required and how different techniques impact the overall efficiency of the processor.

Part A – Conceptual Understanding

1. Core Functions of the I/O Module

The I/O module acts as the "intelligence" between the system bus and external peripherals. To manage these complex interactions, it performs five primary functions:

- **Control and Timing:** The module coordinates the flow of traffic between internal resources (CPU/Memory) and external devices. It ensures that the system bus is not congested by slow peripheral responses, maintaining synchronized communication.
- **Processor Communication:** The module must "talk" to the CPU by decoding commands such as READ, WRITE, TEST, and CONTROL. It also reports status signals, such as BUSY or READY, and exchanges data via the data bus.
- **Device Communication:** It facilitates communication with the peripheral by sending specific device-level commands and receiving status signaling tailored to that specific hardware.
- **Data Buffering:** This is essential for handling the **"Speed Mismatch"**. While the CPU operates at nanosecond scales, peripherals often operate at millisecond scales. The module acts as a temporary reservoir, holding data until either the CPU or the device is ready to process it.
- **Error Detection:** The module monitors for mechanical malfunctions (e.g., paper jams) or electrical transmission errors (e.g., bit changes) and reports these issues back to the processor.

2. Comparison of Performance Techniques and "Processor Paralysis"

The evolution of I/O techniques is primarily a journey toward reducing the workload on the CPU.

- **Programmed I/O:** In this mode, the CPU is responsible for every step of the transfer. The processor enters a "busy-waiting" loop, repeatedly checking the status bit of the I/O module. As emphasized in the lecture, this leads to **"Processor Paralysis"**. Because the CPU is stuck in this loop, it is unable to perform other useful calculations, such as X + Y, effectively wasting its high-speed processing power on "waiting" for a slow device.
- **Interrupt-Driven I/O:** This technique seeks to **"free"** the CPU from the busy-waiting loop. The CPU issues a command and immediately moves on to execute other instructions. The I/O module "pokes" the CPU via an interrupt signal only when the data is ready for transfer.
- **Direct Memory Access (DMA):** DMA represents the ultimate offloading of I/O tasks. A specialized hardware chip takes control of the system bus to move blocks of data directly between memory and the device. The CPU is only involved at the very beginning to set up the transfer and at the end to acknowledge completion.

3. Addressing Modes: Memory-Mapped vs. Isolated I/O

Architectures differ in how they distinguish between memory addresses and I/O addresses.

Feature	Memory-Mapped I/O	Isolated I/O
Address Space	Devices and memory share a single, unified address space.	I/O devices have a dedicated, separate address space from memory.
Instruction Set	Uses standard memory instructions like Load and Store.	Requires specialized Instructions such as IN and OUT.
Hardware Complexity	Simpler bus design; requires only one read/write control line.	More complex; needs extra “I/O Select” lines to differentiate the spaces.
Impact on Memory	“Steals” or consumes part of the available memory address range.	Keeps the entire memory address space available for the OS and applications.

Part B – Interrupt Handling Analysis

1. Resolving the "Two Design Issues"

When implementing an interrupt-driven system, two fundamental design challenges arise that the processor must address to maintain system integrity:

- **Device Identification:** How does the processor determine which specific device issued the interrupt among multiple I/O modules?
- **Priority Determination:** If multiple interrupts occur simultaneously, how does the processor decide which one to service first?

To resolve these, modern architectures utilize four primary categories of techniques: **Multiple Interrupt Lines**, **Software Polling**, **Daisy Chaining**, and **Bus Arbitration**.

2. Device Identification Techniques

- **Software Poll:** When the processor detects an interrupt, it branches to an Interrupt Service Routine (ISR) that polls each I/O module sequentially. As discussed in our lectures, this is akin to "knocking on every door" to find the source. While simple, it is highly time-consuming and inefficient for systems with many peripherals.
- **Daisy Chaining (Hardware Poll):** This provides a more efficient hardware-based polling mechanism. An "Interrupt Acknowledge" signal propagates through the modules in a serial fashion. The first requesting module in the chain intercepts this signal and responds by placing a unique "Vector" (ID) on the data lines. This **Vectored Interrupt** allows the CPU to jump directly to the specific device's service routine.

- **Bus Arbitration:** In high-performance systems, an I/O module must first gain control of the system bus before it can raise an interrupt request. This ensures that only one identified device is serviced at any given time, providing excellent scalability.

3. Priority Logic: The "Fire Sensor" Analogy

As emphasized in the lecture, priority in a Daisy Chain is determined by the physical ordering of the devices. The device physically closest to the processor on the "Interrupt Acknowledge" line has the highest priority.

Case Study: Consider a system with a **Fire Sensor** and a **Lamp Switch**.

- If both devices request an interrupt at the same time, the processor must prioritize the life-critical task.
- By placing the Fire Sensor at the beginning of the Daisy Chain, the hardware ensures it "grabs" the acknowledge signal first.
- This physical prioritization ensures that the CPU handles the fire alert immediately, while the less critical "lamp" request waits until the higher-priority task is complete.

4. Technical Comparison and Recommendations

Technique	Latency	Hardware Complexity	Scalability
Software Poll	High (Slowest)	Very Low	Poor
Daisy Chaining	Medium	Medium	Good
Bus Arbitration	Low (Fastest)	High	Excellent

Recommendation: For **Embedded Systems** where cost and simplicity are paramount, **Daisy Chaining** is recommended due to its balance of speed and low hardware overhead. However, for **High-Performance Servers**, **Bus Arbitration** is necessary to manage massive traffic from multiple high-speed cores without overwhelming the processor.

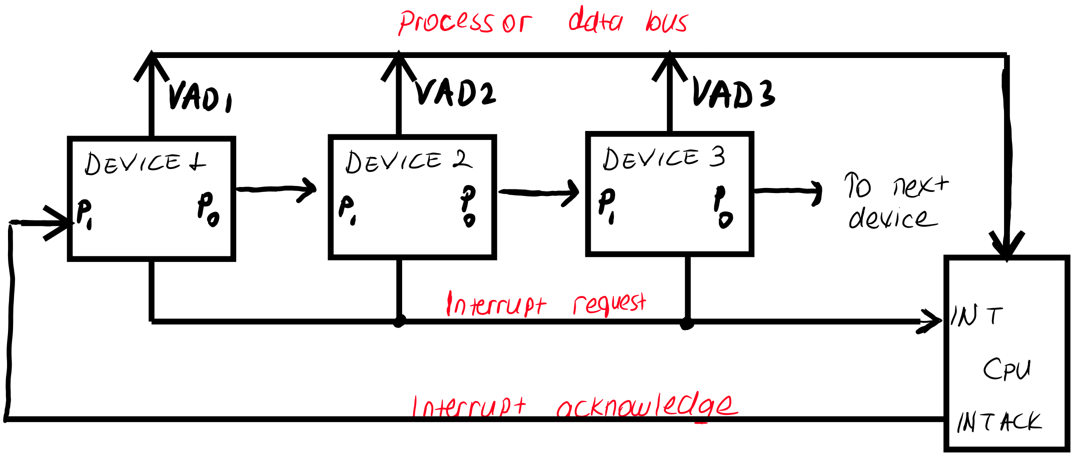

Figure 1: Daisy-Chain Priority Interrupt Logic

This section provides a quantitative and qualitative analysis of I/O performance. We will examine why **Direct Memory Access (DMA)** is the superior choice for high-volume data movement by comparing the clock cycle consumption of three fundamental techniques.

Part C – Performance Comparison Scenario

1. Quantitative Analysis: Efficiency Calculations

To illustrate the performance gap, we analyze a scenario involving a **1 MB** data transfer (approximately 250,000 4-byte words) on a system with a **3 GHz** processor. We assume the system generates one interrupt for every 1 KB of data transferred.

- Programmed I/O: The CPU manages every single word transfer, consuming an average of 100 cycles per word.

  Total Cycles = 250,000 words × 100 cycles/word = **25,000,000 cycles**

- Interrupt-Driven I/O: The CPU is only involved when a 1 KB block is ready, requiring 500 cycles to service each interrupt.

  Total Cycles = 1,000 interrupts × 500 cycles/interrupt = **500,000 cycles**

- Direct Memory Access (DMA): The processor is only involved during the initial setup and final termination. The data movement itself is handled by the DMAC.

  Total Cycles (Overhead) = Setup + Termination = **2,000 cycles**

2. Qualitative Analysis: Context Switching vs. Cycle Stealing

The drastic difference in cycle consumption is not merely numerical; it stems from the fundamental way the processor interacts with the I/O module.

- **The Overhead of Context Switching:** In Interrupt-Drven I/O, the processor must perform a full **Context Switch** for every request. This involves freezing the current process and saving the **Program Status Word (PSW)** and **Program Counter (PC)** to the control stack in main memory. This "administrative" work consumes the majority of the 500-cycle service time.
- **The Efficiency of Cycle Stealing:** DMA utilizes a mechanism called **Cycle Stealing**. Instead of interrupting the processor and forcing a state save, the DMA controller simply "steals" a bus cycle when the processor is not using it (e.g., during a decode stage or between instruction cycles).
- **The Key Distinction:** As emphasized in the lecture, DMA **does not save the PSW** or PC. The processor merely pauses for one cycle and then resumes immediately, making DMA approximately **12,500 times more efficient** than Programmed I/O in this scenario.

3. Comparative Efficiency Summary

Technique	Processor Involvement	Context Saving Required?	Total Processor Overhead
Programmed I/O	Constant (Busy-waiting)	No	25,000,000 Cycles
Interrupt-Driven	Moderate (Service routines)	**Yes (PSW/PC saving)**	500,000 Cycles
DMA	Minimal (Setup/End only)	**No (Cycle Stealing)**	2,000 Cycles

4. Scaling and Efficiency Analysis

Based on the quantitative results derived from the 1 MB transfer scenario, we can conclude the following regarding the scalability and efficiency of I/O mechanisms:

- **Best Scaling: DMA scales best** among the three techniques. This is because the processor overhead for DMA (2,000 cycles) remains constant for the entire data block, regardless of its size. In contrast, the costs for Programmed I/O and Interrupt-Driven I/O increase linearly with the amount of data transferred, making them unsuitable for large-scale operations.
- **Most Efficient: DMA is the most CPU-efficient technique**. In this 1 MB scenario, DMA consumes only **0.008%**of the cycles that would be required by Programmed I/O (2,000/25,000,000), representing a massive reduction in processor involvement.
- **Modern Preference:** DMA is preferred in modern high-performance systems because it eliminates the repetitive administrative burden of **Context Switching** and **PSW saving**. By offloading the **intensive data movement tasks** to the DMA controller, the processor is freed to handle high-level computational applications while I/O operations occur autonomously in the background.

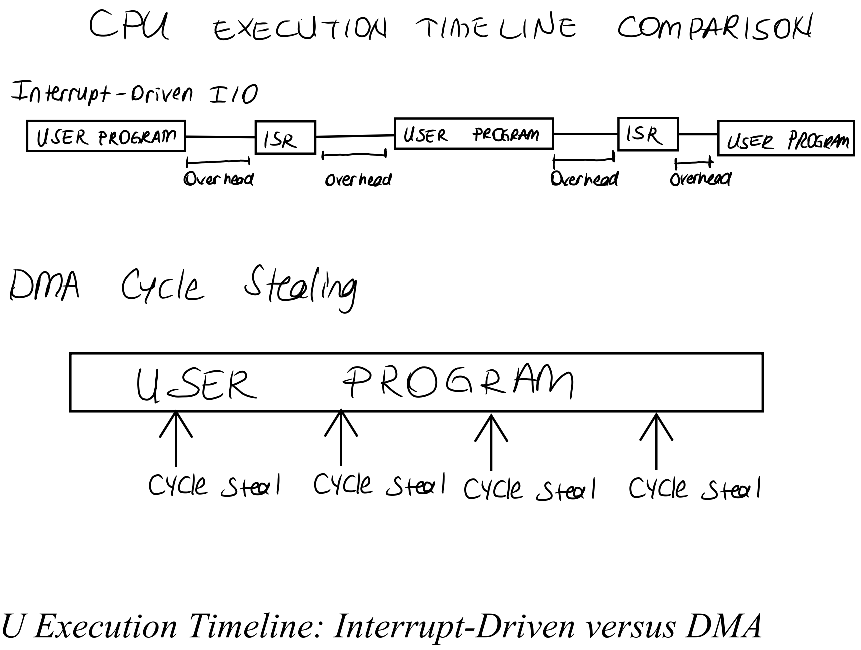

Figure 2: CPU Execution Timeline: Interrupt-Driven versus DMA

Part D – Modern I/O Technology: PCI Express (PCIe)

As computer architectures evolved to handle massive amounts of data, traditional shared-bus systems became a significant bottleneck. The emergence of **PCI Express (PCIe)** revolutionized internal communication by moving away from shared architectures to a more scalable and efficient design.

1. Switch-Based Point-to-Point Topology

Unlike older bus standards where all devices shared the same electrical pathway, PCIe utilizes a **switch-based point-to-point** topology. In this architecture, each device has a dedicated, bidirectional link (lane) to a central switch. This design eliminates bus contention, allowing multiple high-speed devices to transmit data simultaneously without waiting for the bus to become idle. This is essential for modern hardware such as **NVMe SSDs** and high-end **GPUs**, which require dedicated bandwidth to function at peak performance.

2. Data Rates and Performance Evolution

The performance of PCIe is measured in Gigatransfers per second (GT/s). Modern iterations have seen a dramatic increase in throughput:

- **PCIe 5.0/6.0:** These versions can reach speeds between **32 to 64 GT/s per lane**.
- **Aggregate Bandwidth:** In a typical x16 configuration (common for GPUs), the total bandwidth is sufficient to satisfy even the most demanding AI workloads and data center requirements.

3. Typical Use Cases and Modern Workload Suitability

PCIe serves diverse high-performance applications: **Graphics cards** (NVIDIA RTX 4090) use x16 lanes for AI training and real-time rendering; **NVMe SSDs** (Samsung 990 PRO) achieve 7+ GB/s for low-latency storage; **Network adapters** (10/100 GbE NICs) enable data center connectivity; **Video capture cards** handle uncompressed 4K/8K workflows; and **ML accelerators** (TPUs, FPGAs) require direct memory access for neural network processing. The dedicated point-to-point topology and scalable bandwidth make PCIe ideal for concurrent, high-throughput workloads where bus contention would create bottlenecks.

4. Integration with the Ring Interconnect System

A critical advancement in modern processor architecture (specifically in high-end server chips like Intel Xeon) is the integration of I/O agents directly into the **Ring Interconnect System**.

- **Mechanism:** The ring is a high-speed, bidirectional communication path that connects all CPU cores, the L3 cache, and I/O agents (like PCIe and QPI).
- **Ring Agents:** Each component on the ring acts as a **ring agent**, utilizing a distributed protocol to allocate time slots for data transfer.
- **Efficiency:** This allows data from a PCIe device to travel in the shortest direction along the ring to reach its destination (either a specific core or the cache) with minimal latency.

Consequently, **CPU involvement is minimized** as the PCIe controllers and Ring Agents manage data routing autonomously, allowing the processor cores to remain focused on computational tasks.

This final section of the report addresses the modern challenges of high-speed networking and the architectural solutions developed to overcome the limitations of traditional Direct Memory Access (DMA).

Part E – Critical Thinking: Enhancing I/O Performance with Direct Cache Access (DCA)

1. The Limitations of Traditional DMA in High-Speed Environments

While DMA was a revolutionary improvement over interrupt-driven I/O, modern network speeds—ranging from **10 to 100 Gbps**—have exposed new bottlenecks. Traditional DMA operations face three primary challenges:

- **Memory Access Bottleneck:** Memory speeds (latency and bandwidth) have not kept pace with the rapid advances in CPU and network speeds. Even fast DRAM (DDR4/DDR5) cannot always match the rate at which data is received by a high-speed Network Interface Card (NIC).
- **Cache Inefficiency and Invalidation:** When the DMA controller writes data directly to main memory, it triggers the **Inclusion Property**, which forces the CPU to invalidate the corresponding lines in its L3 cache to maintain coherence. Consequently, when the application needs to process this data, the CPU suffers a "Cache Miss" and must perform a slow off-chip read from RAM.
- **Processor Idle Time:** The CPU often waits for memory operations to complete, leading to significant inefficiencies during high-speed packet processing.

2. The Solution: Direct Cache Access (DCA)

To solve these bottlenecks, modern architectures (such as Intel Xeon with DDIO) utilize **Direct Cache Access (DCA)**. DCA allows I/O operations to bypass main memory and interact directly with the **Last-Level Cache (L3)**.

- **Cache Injection:** In an input operation, the I/O controller writes data directly into the L3 cache rather than RAM. This "Cache Injection" ensures that the CPU finds the data "at its doorstep," eliminating the need for slow memory fetches.
- **Output Optimization:** For output operations, the Memory Controller Hub (MCH) checks if the data resides in the L3 cache first. If it is present, the data is sent directly to the I/O controller, completely bypassing the main memory.
- **Performance Impact:** DCA minimizes cache misses and reduces processor overhead, which is vital for real-time data processing and high-concurrency server environments.

3. Conclusion: The Evolution of the I/O Function

The evolution of I/O reflects a continuous effort to reduce CPU involvement and increase autonomy:

1. **Direct CPU Control:** The CPU manages the device directly.
2. **I/O Module (Programmed I/O):** A controller is added, but the CPU still waits.
3. **Interrupt-Driven I/O:** CPU is freed until the device is ready.
4. **DMA:** I/O module moves data blocks directly to memory.
5. **I/O Channel:** The module executes its own specialized I/O programs.
6. **I/O Processor:** The module becomes a fully independent processor with its own local memory.

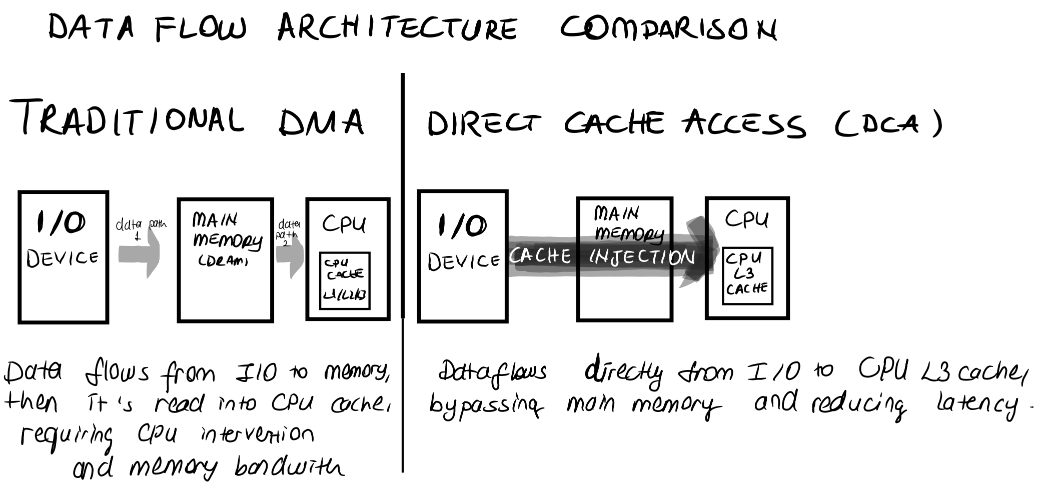
DATA FLOW ARCHITECTURE COMPARISON
TRADITIONAL DMA
DIRECT CACHE ACCESS (DCA)
I/O DEVICE
data path 1
MAIN MEMORY (DRAM)
data path 2
CPU
CPU CACHE L1/L2/L3
I/O DEVICE
CACHE INJECTION
MAIN MEMORY
CPU
CPU L3 CACHE
Data flows from I/O to memory, then It's read into CPU cache, requiring CPU intervention and memory bandwith
Dataflows directly from I/O to CPU L3 cache, bypassing main memory and reducing latency.

Figure 3: Data Flow Architecture: Traditional DMA vs. Direct Cache Access (DCA)

Conclusion: The Future of I/O Autonomy

The evolution of I/O mechanisms, as analyzed in this report, reflects a relentless drive toward greater architectural autonomy and the reduction of processor overhead. We have transitioned from the era of **Programmed I/O**, where the CPU suffered from total **"Processor Paralysis"**, to the sophisticated era of **Direct Cache Access (DCA)**.

Our analysis proves that while **Interrupt-Driven I/O** freed the CPU from busy-waiting, the administrative burden of **Context Switching** (saving the PSW and PC) remained a significant performance tax. The introduction of **Direct Memory Access (DMA)** was a pivotal turning point, replacing heavy context saves with efficient **Cycle Stealing**. However, as networking speeds reached the 100 Gbps threshold, even DMA faced the "Memory Bottleneck," leading to the modern necessity of **Cache Injection** through technologies like Intel DDIO.

In conclusion, the design of I/O mechanisms is no longer just about moving data; it is about managing **latency** and **cache coherence** in a multi-core environment. As we look toward the future, the I/O module continues to evolve into a fully independent **I/O Processor**, ensuring that the CPU remains dedicated to high-level computation while the I/O subsystem handles the increasing data demands of the modern world.

References

[1] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 10th ed. Hoboken, NJ: Pearson Education, 2016.

[2] Intel Corporation, "Intel® Data Direct I/O Technology (Intel® DDIO): A Guide to Modern I/O Optimization," Technical Whitepaper, 2012.

[3] Course Instructor, "Lecture 7: Input/Output Systems and Bus Structures," Course Materials, Computer Architecture Course, Istanbul Health and Technology University, Fall 2025.