Input-Output Organization: Programmed I/O, Interrupts, and Direct Memory Access

Based on Carl Hamacher, Computer Organization (6th Edition)

August 28, 2025

Outline

Introduction to I/O Organization

- The I/O subsystem connects the CPU and memory with the external environment, enabling interaction with devices like keyboards, monitors, disks, and networks.
- The main challenges addressed by I/O organization:
  - Speed mismatch between fast CPU and slow devices.
  - Different data formats between CPU (binary words) and devices (characters, signals).
  - Managing multiple devices simultaneously.
- Efficient I/O organization is critical for overall system performance.

Types of I/O Devices

- Input Devices: keyboard, mouse, scanner – provide data to CPU.
- Output Devices: monitor, printer, speaker – receive processed results from CPU.
- Storage Devices: HDD, SSD, USB drives – allow long-term data storage.
- Communication Devices: Network Interface Cards (NICs), modems – connect system to networks.
- Each type of device requires specialized handling due to speed and data format differences.

I/O Transfer Techniques

- Program-controlled I/O: CPU executes explicit instructions to monitor devices and transfer data.
- Interrupt-driven I/O: Device notifies CPU when ready, reducing CPU idle time.
- Direct Memory Access (DMA): Transfers large blocks directly between device and memory with minimal CPU intervention.

Synchronous vs Asynchronous I/O

- **Synchronous I/O:**
  - CPU waits for device to complete operation.
  - Simple but inefficient if device is slow.
- **Asynchronous I/O:**
  - CPU and device work independently.
  - Device signals CPU (via interrupt) when finished.
  - Increases CPU utilization.
- Handshaking protocols ensure correct timing between CPU and I/O device.

I/O Interface Components

- **Data Register:** Holds data being transferred.
- **Status Register:** Indicates device state (ready, busy, error).
- **Control Register:** Used by CPU to issue commands to the device.
- Control logic translates CPU commands into signals devices can understand.
- Together, these registers form the I/O interface.

Memory-Mapped vs I/O-Mapped I/O

- **I/O Mapped I/O:**
  - Separate instructions for I/O (IN, OUT).
  - Limited address space for devices.
- **Memory-Mapped I/O:**
  - Device registers treated as memory addresses.
  - Uniform instruction set for memory and I/O.
  - Simplifies CPU design but reduces memory address space.

Handshaking in I/O

- Handshaking ensures synchronization between CPU and slower devices.
- Example signals:
  - READY/BUSY – device signals readiness.
  - ACK/NACK – CPU acknowledges data.
- Prevents data loss and ensures proper timing.
- Example: A printer signals READY when buffer is available for next character.

Typical I/O Transaction Cycle

1. CPU writes command into device control register.
2. Device executes operation and updates status register.
3. CPU polls status register or waits for interrupt.
4. Data transfer occurs via data register.

This sequence applies to both input and output devices.

Problems with CPU-Controlled I/O

- CPU must frequently check device status.
- Leads to busy waiting and poor utilization.
- Cannot keep up with high-speed I/O (like disks or networks).
- Motivates use of interrupts and DMA.

Section 8.1 Summary

- I/O subsystem bridges CPU and peripherals with major speed and format differences.
- Interfaces provide data, control, and status registers.
- Program-controlled I/O is inefficient, leading to advanced methods like interrupts and DMA.

Concept of Program-Controlled I/O

- CPU takes full responsibility for monitoring device readiness.
- Data transfers are initiated only when device indicates ready in its status register.
- Example: Keyboard input – CPU checks the READY bit before reading data.

Working of Programmed I/O

- The CPU executes instructions to manage I/O transfers step by step.
- Each transfer involves:
  1. An **input instruction** to move data from device $\rightarrow$ CPU register.
  2. A **store instruction** to move data from CPU register $\rightarrow$ memory.
- No direct path exists between the I/O device and main memory.

CPU Involvement in Programmed I/O

- The CPU stays in a **program loop** (busy waiting).
- It repeatedly checks the device status register until the I/O device signals ready.
- This wastes valuable CPU cycles since no useful computation is done in the meantime.
- Example: Printer taking milliseconds while CPU operates in nanoseconds.

Instruction Sequence Example

Example sequence for reading input:

1. Read status register.
2. If READY bit $= 0$, repeat.
3. If READY bit $= 1$, read data register.
4. Transfer data to memory or CPU register.

This cycle repeats for each word, making it inefficient for bulk transfers.

Register Transfer Notation

- Example program fragment:
  - MOV R1, STATUS (load device status)
  - TEST R1, READY (check ready bit)
  - JZ WAIT (loop if not ready)
  - IN R2, DATA (read data)
  - STORE R2, MEM[addr] (write to memory)
- Every word transfer requires CPU intervention.

Limitations of Program-Controlled I/O

- CPU utilization is extremely low.
- Only practical for very slow devices or occasional transfers.
- Cannot handle simultaneous I/O from multiple devices.
- Example: Network card generating thousands of packets per second.

Flowchart Representation

Flow of program-controlled I/O:

1. Poll device status.
2. If not ready, repeat loop.
3. If ready, perform transfer.
4. Return to step 1 for next word.

Very inefficient but simple to implement.

Performance Issues

- Modern CPUs operate at GHz, I/O devices at kHz–MHz.
- Polling wastes millions of cycles.
- Performance gap widens with faster CPUs.
- Necessitates more advanced approaches.

Section 8.2 Summary

Program-controlled I/O is simple but leads to poor CPU efficiency. Better methods are interrupts and DMA.

Introduction to Interrupts

- Interrupts allow devices to signal CPU when they are ready.
- CPU can execute other tasks instead of waiting.
- Provides asynchronous handling of I/O events.

Interrupt-Initiated I/O: Concept

- Unlike programmed I/O, the CPU does **not** constantly poll for device readiness.
- The I/O interface itself monitors peripherals and sends an **interrupt request** when data is ready.
- The CPU continues executing other tasks until interrupted.

Sequence of Interrupt-I/O Operation

1. CPU issues a command to prepare for I/O and then proceeds with other tasks.
2. The I/O interface continuously monitors device status.
3. When the device is ready, the interface sends an **interrupt request (IRQ)**.
4. CPU suspends current execution, jumps to an Interrupt Service Routine (ISR).
5. ISR performs the I/O transfer and then control returns to the original program.

Advantages of Interrupt-I/O

- CPU doesn't waste cycles polling devices.
- Improves system efficiency and responsiveness.
- Enables better overlap of computation and I/O.

Limitations of Interrupt-I/O

- Transfer rate limited by CPU's ability to service interrupts.
- Each I/O operation still requires CPU involvement through ISR.
- Not as efficient as DMA for large block transfers.

Types of Interrupts (Part 1)

- Hardware Interrupts: Generated by I/O devices through hardware lines.
- Software Interrupts: Triggered by instructions within the program.
- Vectored Interrupts: Predefined address of ISR.
- Non-vectored Interrupts: CPU must determine ISR dynamically.

Types of Interrupts (Part 2)

- Maskable Interrupts: Can be enabled or disabled by the CPU.
- Non-maskable Interrupts (NMI): Cannot be disabled; highest priority.
- External Interrupts: Generated by hardware devices.
- Internal Interrupts: Generated internally by CPU (faults, exceptions).
- Synchronous vs Asynchronous Interrupts: Predictable (timer) vs unpredictable (I/O).

Section 8.3 Summary

- Interrupts eliminate busy waiting and improve CPU utilization.
- Interrupt types allow flexible handling of I/O and system events.
- For high-speed bulk data, DMA is more efficient than interrupt-driven I/O.