Here's a point-by-point explanation of the code implementation for **Week 5: Memory Management** and **Week 6: I/O Operations**:

Week 5: Memory Management

1. **Simulated Memory Space**:
    a. The `MEMORY_SIZE` variable defines the total size of the memory (1024 bytes in this example).
    b. A list, `memory`, represents the simulated memory, initialized with zeros to mimic an empty memory state.
2. **Memory Read/Write Operations**:
    a. **`read_memory(address)`**:
        i. Takes an address as input.
        ii. Checks if the address is valid (within 0 and `MEMORY_SIZE - 1`).
        iii. Returns the value stored at the given memory address.
    b. **`write_memory(address, data)`**:
        i. Takes an address and data as inputs.
        ii. Validates the address.
        iii. Writes the data to the specified address in memory.
        iv. If the address is invalid in either function, it raises an error.
3. **Address Mapping and Memory Segmentation**:
    a. **Segments**:
        i. Memory is divided into three predefined segments:
            1. `"code"`: From 0 to 255.
            2. `"data"`: From 256 to 511.
            3. `"stack"`: From 512 to 1023.
        ii. These segments are stored in the `segments` dictionary with base and limit values.
    b. **`get_physical_address(segment, logical_address)`**:
        i. Converts a logical address within a segment into a physical memory address.
        ii. Validates that the logical address is within the bounds of the segment.
        iii. Calculates the physical address by adding the segment's base to the logical address.

        iv. Raises an error if the segment or address is invalid.

Week 6: I/O Operations

1. **Simulated I/O Devices**:
    a. The `io_devices` dictionary represents simulated devices:
        i. `"keyboard"`: Stores inputs from the user.
        ii. `"display"`: Stores outputs intended for display.
2. **I/O Write Operation**:
    a. **`io_write(device, data)`**:
        i. Takes a device name and data as input.
        ii. Checks if the device exists in the `io_devices` dictionary.
        iii. Appends the data to the specified device's list.
        iv. Raises an error if the device is invalid.
3. **I/O Read Operation**:
    a. **`io_read(device)`**:
        i. Takes a device name as input.
        ii. Checks if the device exists.
        iii. If there is data in the device's list, it returns and removes the first item (FIFO operation).
        iv. Returns None if no data is available in the device.
        v. Raises an error if the device is invalid.
4. **I/O Instruction Execution**:
    a. **`execute_io_instruction(instruction, device, data=None)`**:
        i. Processes I/O operations based on an instruction (`"write"` or `"read"`).
        ii. `"write"`:
            1. Calls `io_write` to send data to the specified device.
        iii. `"read"`:
            1. Calls `io_read` to retrieve data from the device.
        iv. Raises an error if the instruction is invalid.

Testing the Implementation

1. **Memory Management Test**:
    a. Writes the value 42 to the logical address `10` in the `"data"` segment.

    b. Converts the logical address into a physical address using `get_physical_address`.
    c. Reads the value back from memory to ensure proper storage and retrieval.
2. **I/O Operations Test**:
    a. Writes `"Input from user"` to the `"keyboard"` device using `execute_io_instruction`.
    b. Reads the value back from the `"keyboard"` to simulate input processing.
    c. Sends `"Hello, World!"` to the `"display"` device.
    d. Verifies the `"display"` contains the output.

This approach modularly handles memory and I/O operations, ensuring scalability for adding new features like paging or more complex I/O instructions later. Let me know if you'd like to expand or refine any part!