SNS COLLEGE OF TECHNOLOGY

(An Autonomous Institution)

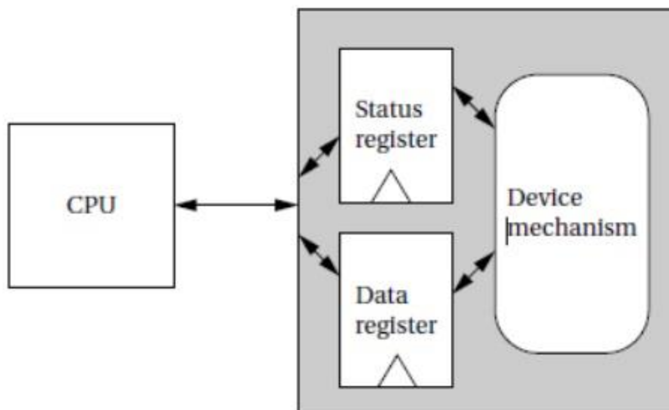
COIMBATORE – 641035

19MCE304- DESIGN OF EMBEDDED SYSTEMS

CPU: Programming input and output:

The basic techniques for I/O programming can be understood relatively independent of the instruction set. In this section, we cover the basics of I/O programming and place them in the contexts of both the ARM and C55x.

We begin by discussing the basic characteristics of I/O devices so that we can understand the requirements they place on programs that communicate with them.



1. Input and Output Devices:

Input and output devices usually have some analog or non electronic component for instance, a disk drive has a rotating disk and analog read/write electronics. But the digital logic in the device that is most closely connected to the CPU very strongly resembles the logic you would expect in any computer system.

Figure 1.18 shows the structure of a typical I/O device and its relationship to the CPU.The interface between the CPU and the device's internals (e.g.,the rotating disk and read/write electronics in a disk drive) is a set of registers. The CPU talks to the device by reading and writing the registers.

Devices typically have several registers:

· *Data registers* hold values that are treated as data by the device, such as the data read or written by a disk.

Status registers provide information about the device's operation, such as whether the current transaction has completed.

Some registers may be read-only, such as a status register that indicates when the device is done, while others may be readable or writable.

2. Input and Output Primitives:

Microprocessors can provide programming support for input and output in two ways: ***I/O instructions*** and ***memory-mapped I/O***.

Some architectures, such as the Intel x86, provide special instructions (in and out in the case of the Intel x86) for input and output. These instructions provide a separate address space for I/O devices.

But the most common way to implement I/O is by memory mapping even CPUs that provide I/O instructions can also implement memory-mapped I/O.

As the name implies, memory-mapped I/O provides addresses for the registers in each I/O device. Programs use the CPU's normal read and write instructions to communicate with the devices.

3. Busy-Wait I/O:

The most basic way to use devices in a program is *busy-wait I/O*. Devices are typically slower than the CPU and may require many cycles to complete an operation. If the CPU is performing multiple operations on a single device, such as writing several characters to an output device, then it must wait for one operation to complete before starting the next one. (If we try to start writing the second character before the device has finished with the first one, for example, the device will probably never print the first character.) Asking an I/O device whether it is finished by reading its status register is often called polling.

Programming input and output (I/O) in the context of CPUs involves managing how data is transferred between the CPU and external devices such as keyboards, displays, storage devices, and networks. This interaction is crucial for the overall operation of computer systems. Here are key concepts and methods related to programming I/O for CPUs:

1. Types of I/O

a. Programmed I/O (Polling)

- **Concept**: The CPU actively checks the status of an I/O device at regular intervals (polling) to see if it needs to read or write data.
- **Advantages**: Simple to implement; good for devices that require frequent attention.
- **Disadvantages**: Inefficient, as the CPU wastes cycles checking device status.

b. Interrupt-Driven I/O

- **Concept**: The CPU is interrupted by an external signal when an I/O device needs attention, allowing the CPU to perform other tasks in the meantime.
- **Advantages**: More efficient use of CPU resources compared to polling.
- **Disadvantages**: More complex to implement; handling interrupts requires careful management to ensure data integrity and responsiveness.

c. Direct Memory Access (DMA)

- **Concept**: A DMA controller manages data transfers directly between I/O devices and memory, bypassing the CPU.
- **Advantages**: Frees the CPU from handling large data transfers; improves system efficiency.
- **Disadvantages**: Requires additional hardware (DMA controller); complexity in managing DMA operations.

2. I/O Addressing

a. Memory-Mapped I/O

- **Concept**: I/O devices are assigned specific memory addresses; the CPU uses standard memory instructions to access these addresses.
- **Advantages**: Simplifies the instruction set; allows for easy integration with the CPU's address space.
- **Disadvantages**: Can consume valuable memory address space; requires careful management to avoid conflicts.

b. Port-Mapped I/O (Isolated I/O)

- **Concept**: I/O devices are assigned specific ports; the CPU uses dedicated I/O instructions to access these ports.
- **Advantages**: Keeps memory address space separate from I/O space; reduces potential conflicts.
- **Disadvantages**: Requires a larger instruction set; can be less intuitive for programming.

3. Programming Interfaces

a. Low-Level Programming (Assembly)

- **Direct Register Access**: Programming involves directly manipulating CPU and I/O device registers.
- **Example**: Using IN and OUT instructions in x86 assembly for port-mapped I/O.

b. High-Level Programming (C/C++)

- **Standard Libraries**: High-level languages provide libraries and functions to abstract low-level I/O operations.
- **Example**: Using fopen, fread, fwrite, and fclose in C for file I/O operations.

4. Operating System Support

a. Device Drivers

- **Concept**: Device drivers are specialized software modules that manage communication between the CPU and I/O devices.
- **Role**: Abstract hardware details and provide a standard interface for applications.
- **Example**: A printer driver translates application requests into printer-specific commands.

b. System Calls

- **Concept**: Operating systems provide system calls that allow applications to perform I/O operations without needing to directly manage hardware.
- **Examples**: read(), write(), open(), and close() system calls in Unix/Linux for file I/O.

5. I/O Examples

a. Polling Example in C (Pseudo-Code)

c
Copy code
```
#define STATUS_REGISTER 0x1234
#define DATA_REGISTER 0x5678
#define DEVICE_READY 0x01

while (!(inb(STATUS_REGISTER) & DEVICE_READY));  // Wait for device to be ready
char data = inb(DATA_REGISTER);  // Read data from device
```

b. Interrupt-Driven I/O Example in C (Pseudo-Code)

c
Copy code
```
void device_isr() {
    char data = inb(DATA_REGISTER);  // Read data from device
    // Process data
}

void setup_interrupt() {
    // Code to enable interrupts and register device_isr as the interrupt handler
}
```

c. DMA Example in C (Pseudo-Code)

c
Copy code
```
void setup_dma() {
    // Configure DMA controller to transfer data from device to memory
    dma_set_source(DEVICE_ADDRESS);
    dma_set_destination(MEMORY_ADDRESS);
    dma_set_transfer_size(SIZE);
    dma_start_transfer();
}

// CPU can perform other tasks while DMA transfer is in progress
```