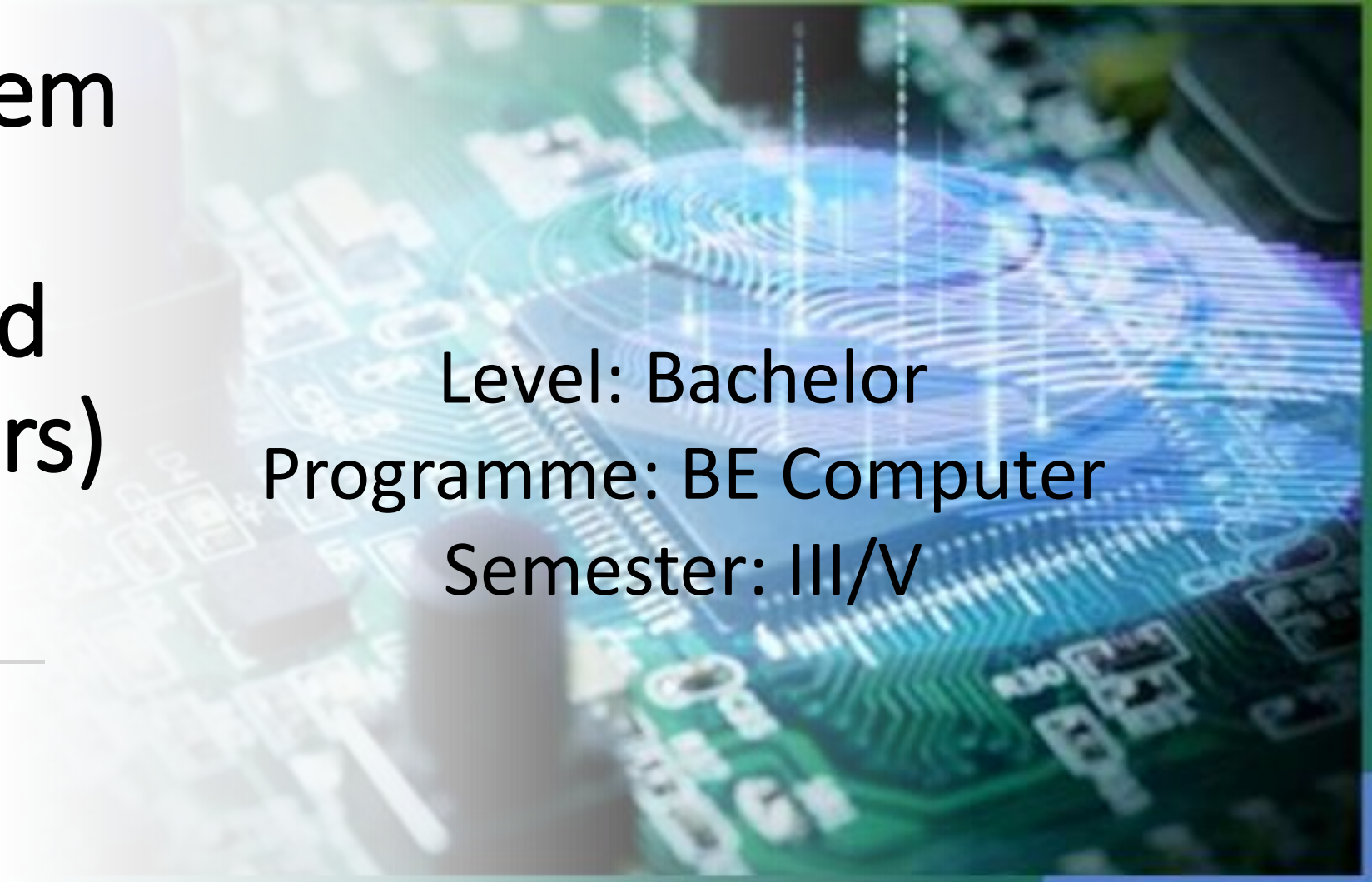


Embedded System

Chapter 6.

Peripherals and Interfacing (4 hrs)



Level: Bachelor
Programme: BE Computer
Semester: III/V

Ishwor Koirala, Ph.D.

Instructor

Outline

Introduction

Books

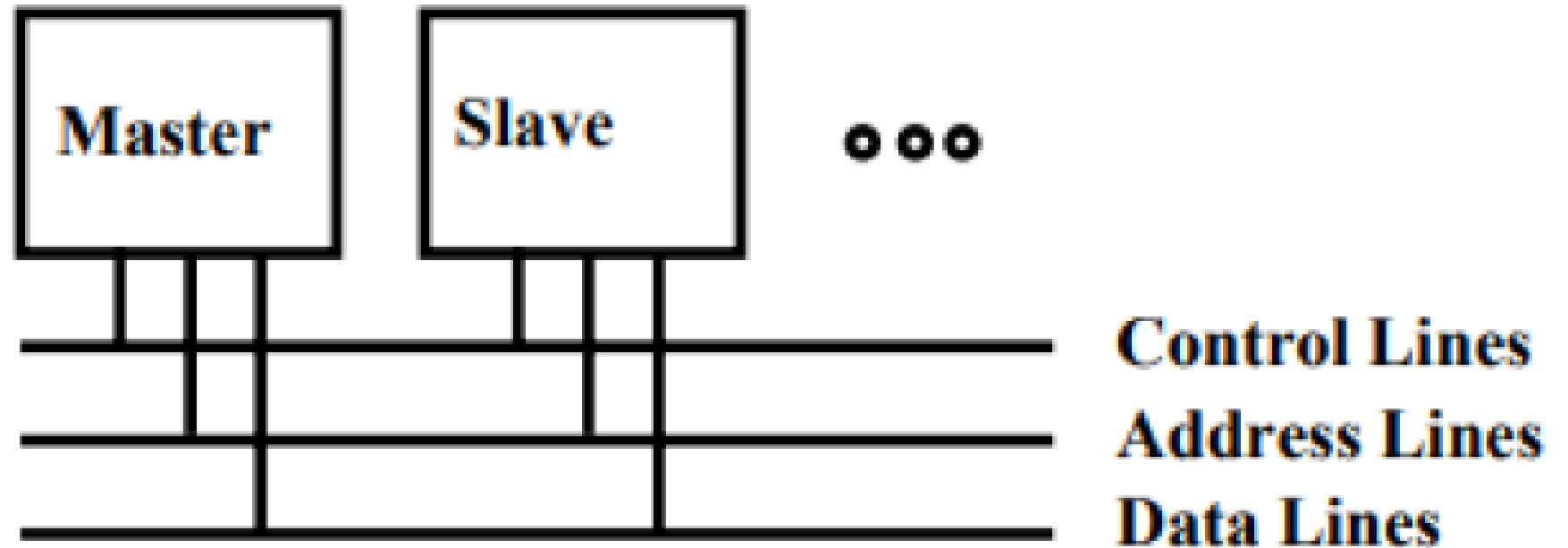
Course Contents

1. Introduction to Embedded system (3 hrs)
2. Programming for Embedded systems (5 hrs)
3. Real-time operating systems (RTOS) (5 hrs)
4. Embedded System Design using VHDL (5 hrs)
5. Communications Protocols (3 hrs)
- 6. Pheripherals and Interfacing (4 hrs)**
7. Internet of Things (IoT) and Embedded system (3hrs)

Interfacing

Interfacing is a way to communicate and transfer information in either way without ending into deadlocks. In our context it is a way of effective communication in real time. This involves:

- Addressing
- Arbitration
- protocol



Interfacing

- Addressing: The data sent by the master over a specified set of lines which enables just the device for which it is meant.
- Protocols: The literal meaning of protocol is a set of rules. Here it is a set of formal rules describing how to transfer data, especially between two devices.
- Arbitration: it specifies the process of generating the logic to select the bus connection over different connected peripherals.

Interfacing

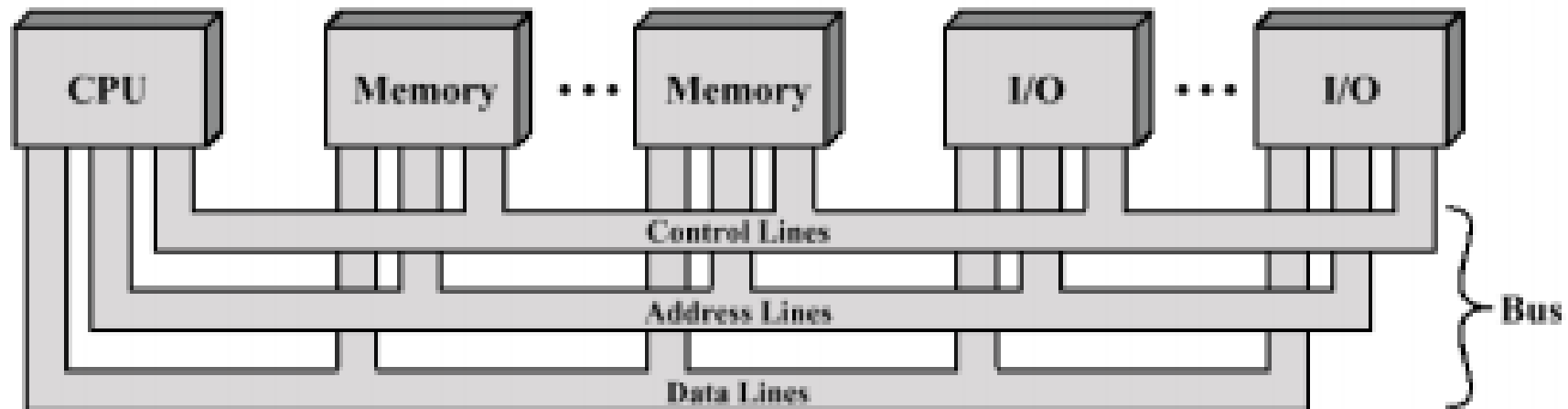
The interface is required:

- For speed synchronization between processor and peripherals since peripherals operate in serial communication mode whereas processor operates in parallel mode.
- To manage the data format since the peripheral operates in bit or byte format where memory and processor operate in word format. The word consists of bytes.
- To maintain the energy of operation since processor and memory use the semiconductor and operate using electronic energy whereas peripherals are electromechanical.

Interfacing

System Buses: Interfacing Processors and Peripherals:

The system bus provides a common path to transfer the data between the two devices i.e. between source and destination. When it is used to transfer the information between different devices within a processor then it is called an internal bus. If they are used to transfer the information between peripherals, processor and memory then called as external bus.



Interfacing

System interconnection structures may support these transfers:

- Memory to processor: the processor reads instructions and data from memory
- Processor to memory: the processor writes data to memory
- I/O to processor: the processor reads data from I/O device
- Processor to I/O: the processor writes data to I/O device
- I/O to or from memory: I/O module allowed to exchange data directly with memory without going through the processor - Direct Memory Access (DMA)

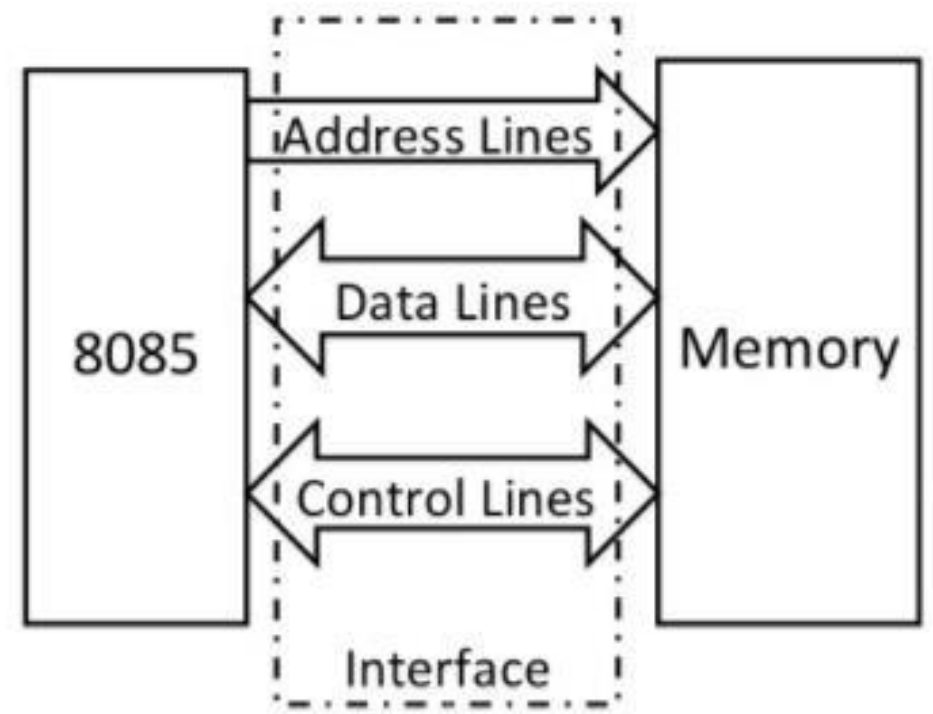
Interfacing

Signals found on a bus are:

- Memory write: data on the bus written into the addressed location
- Memory read: data from the addressed location placed on the bus
- I/O write: data on the bus output to the addressed I/O port
- I/O read: data from the addressed I/O port placed on the bus
- Bus REQ: indicates a module needs to gain control of the bus
- Bus GRANT: indicates that requesting module has been granted control of the bus
- Interrupt REQ: indicates that an interrupt is pending
- Interrupt ACK: Acknowledges that pending interrupt has been recognized
- Reset: Initializes everything connected to the bus
- Clock: on a synchronous bus, everything is synchronized to this signal

Processor and Memory Interfacing:

- Processor and memory are interconnected through the system bus and system bus consists the three different lines as data, control and address.
- A enable signal is use to active the memory for memory operation.
- The devices they participate in communication are called as actor.
- There are two types of actor as: master actor and slave actor.
- The master actor initiates the communication whereas the slave actor response the communication.



Processor and Memory Interfacing:

- Address line provides the unique identification of memory location that stores the word which is going to transfer or write.
- Data lines transfers data between master and slave i.e. it is called as bidirectional.
- Control lines are used to transfer the read/write signals for memory read or write operation.
- Same lines are used to transfer the different values like control, data or address so system bus can be multiplexed. The system bus can act as address bus or data bus or control bus at a time. This method of selecting the bus for particular operation is called **time multiplexing**.
- Different types of control methods are used for communication they are.

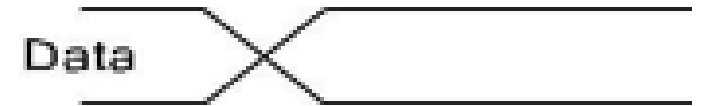
Processor and Memory Interfacing:

- Simple transfer
- Strobe transfer
- Single handshake transfer
- Double handshake transfer

- **Simple Transfer**

For simple transfer, memory and processor transfer the valid data by placing on data bus blindly.

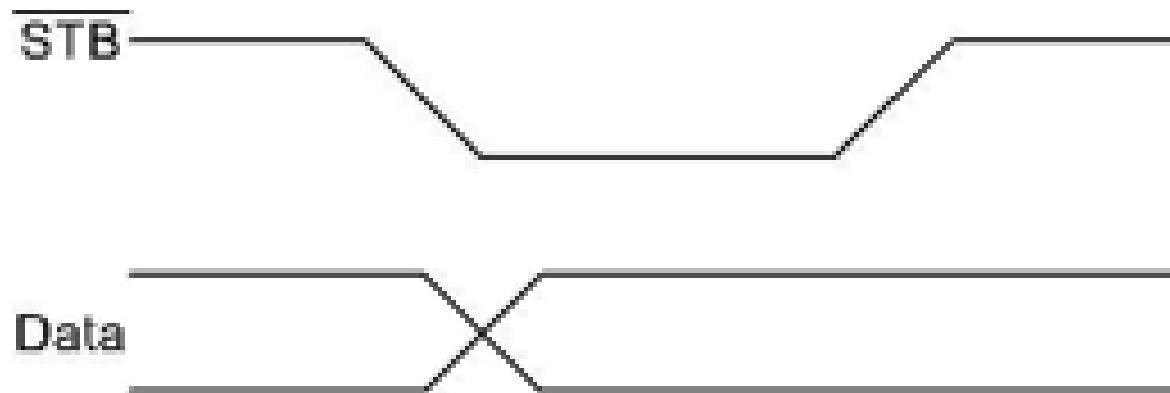
Here cross line indicate the time for new valid data.



Processor and Memory Interfacing:

Strobe Transfer:

In this mode the transmitter transmits the data by placing valid data on data bus and raises the strobe pulse to indicate initiation of data transfer.



Processor and Memory Interfacing:

Single Handshake Transfer:

- The peripheral outputs some data and send signal to processor. —here is the data for you.
- Processor detects asserted signal, reads the data and sends an acknowledge signal (ACK) to indicate data has been read and peripheral can send next data. —I got that one, send me another.
- MP sends or receives data when peripheral is ready.

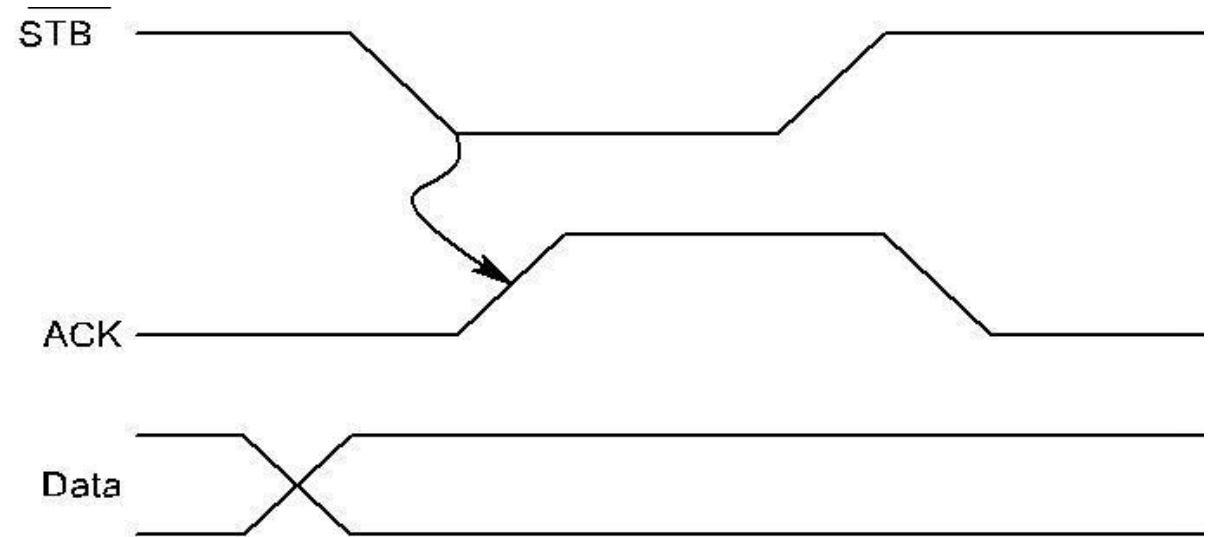
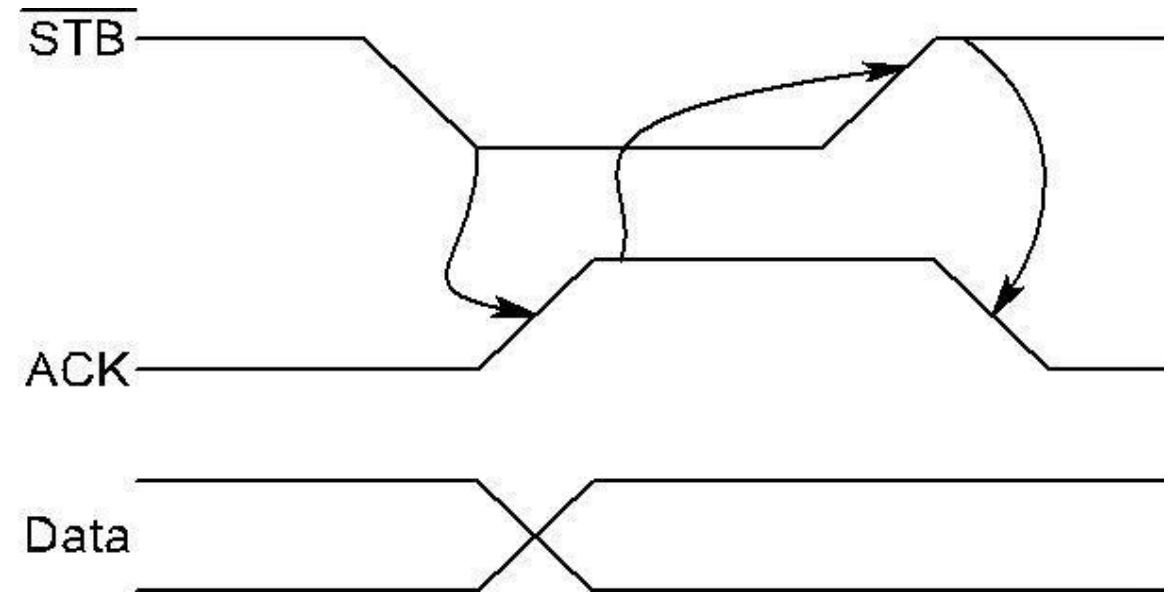


Fig: Single Handshaking

Processor and Memory Interfacing:

Double Handshake Transfer:

- The peripheral asserts its line low to ask processor —Are you ready?
- The processor raises its ACK line high to say — I am ready.
- Peripheral then sends data and raises its line low to say —Here is some valid data for you.
- Processor then reads the data and drops its ACK line to say, —I have the data, thank you, and I await your request to send the next byte of data.



Processor and Memory Interfacing:

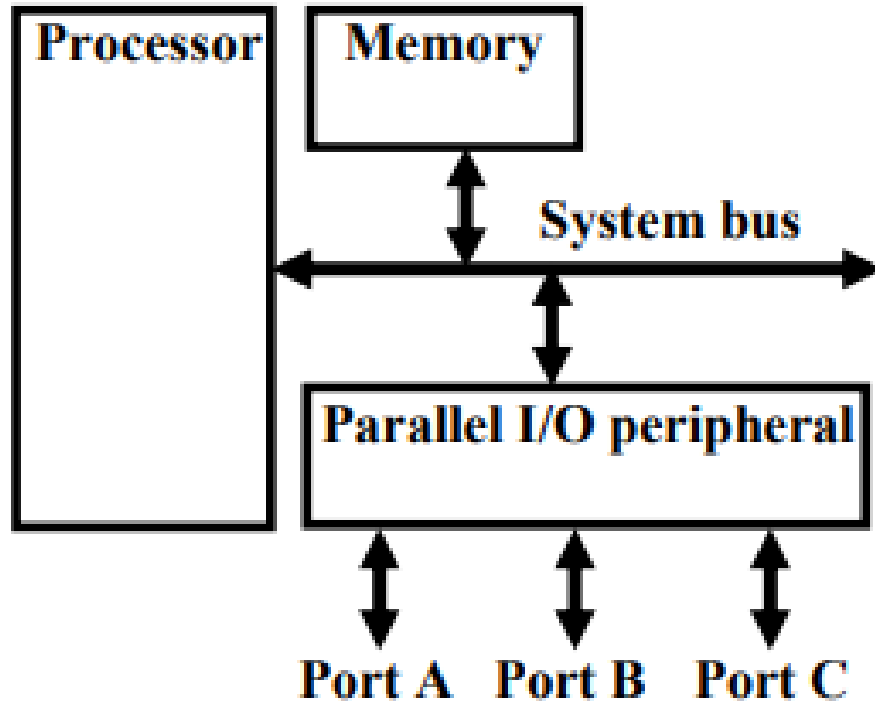
I/O Addressing:

- A microprocessor communicates with other devices using some of its pins. Broadly we can classify them as
- Port-based I/O (parallel I/O)
 - Processor has one or more N-bit ports
 - Processor's software reads and writes a port just like a register
- Bus-based I/O
 - Processor has address, data and control ports that form a single bus
 - Communication protocol is built into the processor
 - A single instruction carries out the read or write protocol on the bus
- Parallel I/O peripheral
 - When processor only supports bus-based I/O but parallel I/O needed.

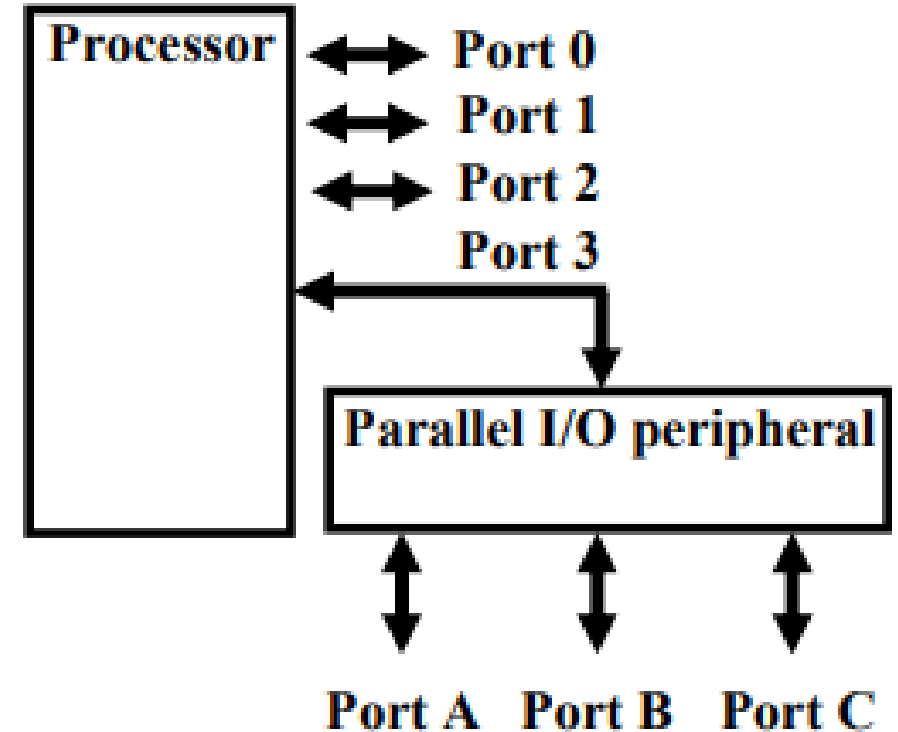
Processor and Memory Interfacing:

- Each port on peripheral connected to a register within peripheral that is read/written by the processor
- Extended parallel I/O
 - When processor supports port-based I/O but more ports needed
 - One or more processor ports interface with parallel I/O peripheral extending total number of ports available for I/O

Processor and Memory Interfacing:



Adding parallel I/O to a bus-based I/O processor



Extended parallel I/O

Processor and Memory Interfacing:

Memory-mapped I/O and standard I/O

- Processor talks to both memory and peripherals using same bus – two ways to talk to peripherals
- Memory-mapped I/O
 - Peripheral registers occupy addresses in same address space as memory
 - e.g., Bus has 16-bit address
 - ✓ lower 32K addresses may correspond to memory
 - ✓ upper 32k addresses may correspond to peripherals
- Standard I/O (I/O-mapped I/O)
 - Additional pin (M/IO) on bus indicates whether a memory or peripheral access
 - e.g., Bus has 16-bit address
 - ✓ all 64K addresses correspond to memory when M/IO set to 0
 - ✓ all 64K addresses correspond to peripherals when M/IO set to 1

Processor and Memory Interfacing:

Memory-mapped I/O vs. Standard I/O

- Memory-mapped I/O
 - Requires no special instructions
 - ✓ Assembly instructions involving memory like MOV and ADD work with peripherals as well
 - ✓ Standard I/O requires special instructions (e.g., IN, OUT) to move data between peripheral registers and memory
- Standard I/O – No loss of memory addresses to peripherals
 - Simpler address decoding logic in peripherals possible
 - When number of peripherals much smaller than address space then high-order address bits can be ignored – smaller and/or faster comparators

Processor and Memory Interfacing:

Address decoding:

- Microprocessor is connected with memory and I/O devices via common address and data bus.
- Only one device can send data at a time and other devices can only receive that data.
- If more than one device sends data at the same time, the data gets garbled.
- In order to avoid this situation, ensuring that the proper device gets addressed at proper time, the technique called address decoding is used.
- In address decoding method, all devices like memory blocks, I/O units etc. are assigned with a specific address.
- The address of the device is determined from the way in which the address lines are used to derive a special device selection signal called chip select (CS).

Processor and Memory Interfacing:

- If the microprocessor has to write or to read from a device, the CS signal to that block should be enabled and the address decoding circuit must ensure that CS signal to other devices are not activated.
- Depending upon the no. of address lines used to generate chip select signal for the device, the address decoding is classified as:

1. I/O mapped I/O

In this method, a device is identified with an 8 bit address and operated by I/O related functions IN and OUT for that $IO/M = 1$.

Since only 8 bit address is used, at most 256 bytes can be identified uniquely. Generally low order address bits A0-A7 are used and upper bits A8-A15 are considered don't care.

Usually I/O mapped I/O is used to map devices like 8255A, 8251A etc.

Processor and Memory Interfacing:

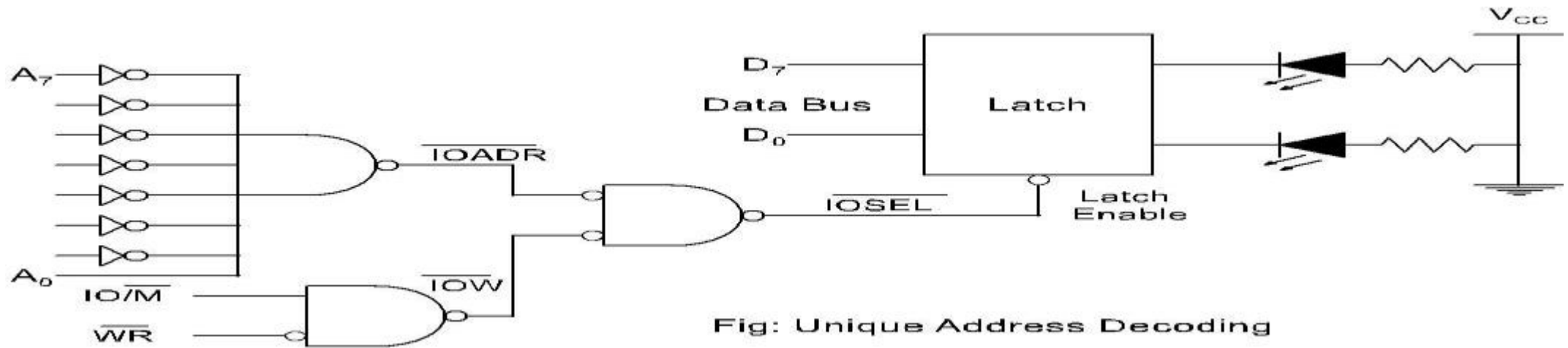
2. Memory mapped I/O

- In this method , a device is identified with 16 bit address and enabled memory related functions such as STA , LDA for which IO/M =0, here chip select signal of each device is derived from 16 bit address lines thus total addressing capability is 64K bytes .
- Usually memory mapped I/O is used to map memories like RAM, ROM etc.
- Depending on the address that are allocated to the device the address decoding are categorized in the following two groups.

Processor and Memory Interfacing:

1. Unique Address Decoding:

If all the address lines on that mapping mode are used for address decoding then that decoding is called unique address decoding. It means all 8-lines in I/O mapped I/O and all 16 lines in memory mapped I/O are used to derive \overline{CS} signal. It is expensive and complicated but fault proof in all cases.



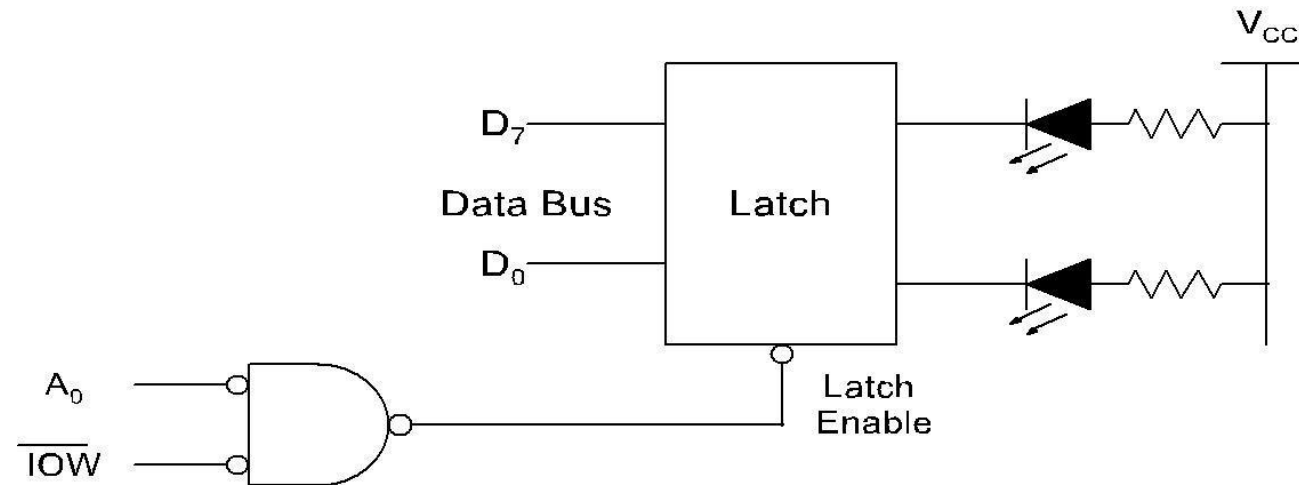
Processor and Memory Interfacing:

2. Non Unique Address decoding:

If all the address lines available on that mode are not used in address decoding then that decoding is called non unique address decoding.

Though it is cheaper there may be a chance of address conflict.

If A_0 is low and \overline{IOW} is low. Then latch gets enabled. Here A_1-A_7 is neglected that is any even address can enable the latch.



Processor and Memory Interfacing:

Interrupts:

- Interrupt is signals send by an external device to the processor, to request the processor to perform a particular task or work.
- Mainly in the microprocessor based system the interrupts are used for data transfer between the peripheral and the microprocessor.
- The processor will check the interrupts always at the 2nd T-state of last machine cycle.
- If there is any interrupt it accept the interrupt and send the INTA (active low) signal to the peripheral.
- The vectored address of particular interrupt is stored in program counter.
- The processor executes an interrupt service routine (ISR) addressed in program counter.
- It returned to main program by RET instruction.

Processor and Memory Interfacing:

Types of Interrupts

Interrupts can be broadly classified as

- Hardware Interrupts
 - These are interrupts caused by the connected devices.
- Software Interrupts
 - These are interrupts deliberately introduced by software instructions to generate user defined exceptions
- Trap
 - These are interrupts used by the processor alone to detect any exception such as divide by zero.

Processor and Memory Interfacing:

Depending on the service the interrupts also can be classified as:

- Fixed interrupt
 - Address of the ISR built into microprocessor, cannot be changed
 - Either ISR stored at address or a jump to actual ISR stored if not enough bytes available
- Vectored interrupt
 - Peripheral must provide the address of the ISR
 - Common when microprocessor has multiple peripherals connected by a system bus

Processor and Memory Interfacing:

- Compromise between fixed and vectored interrupts
 - One interrupt pin
 - Table in memory holding ISR addresses (maybe 256 words)
 - Peripheral doesn't provide ISR address, but rather index into table
 - ✓ Fewer bits are sent by the peripheral
 - ✓ Can move ISR location without changing peripheral

Processor and Memory Interfacing:

Maskable vs. Non-maskable interrupts

- Maskable:
 - programmer can set bit that causes processor to ignore interrupt
 - This is important when the processor is executing a time-critical code
- Non-maskable:
 - a separate interrupt pin that can't be masked
 - Typically reserved for drastic situations, like power failure requiring immediate backup of data to non-volatile memory

Processor and Memory Interfacing:

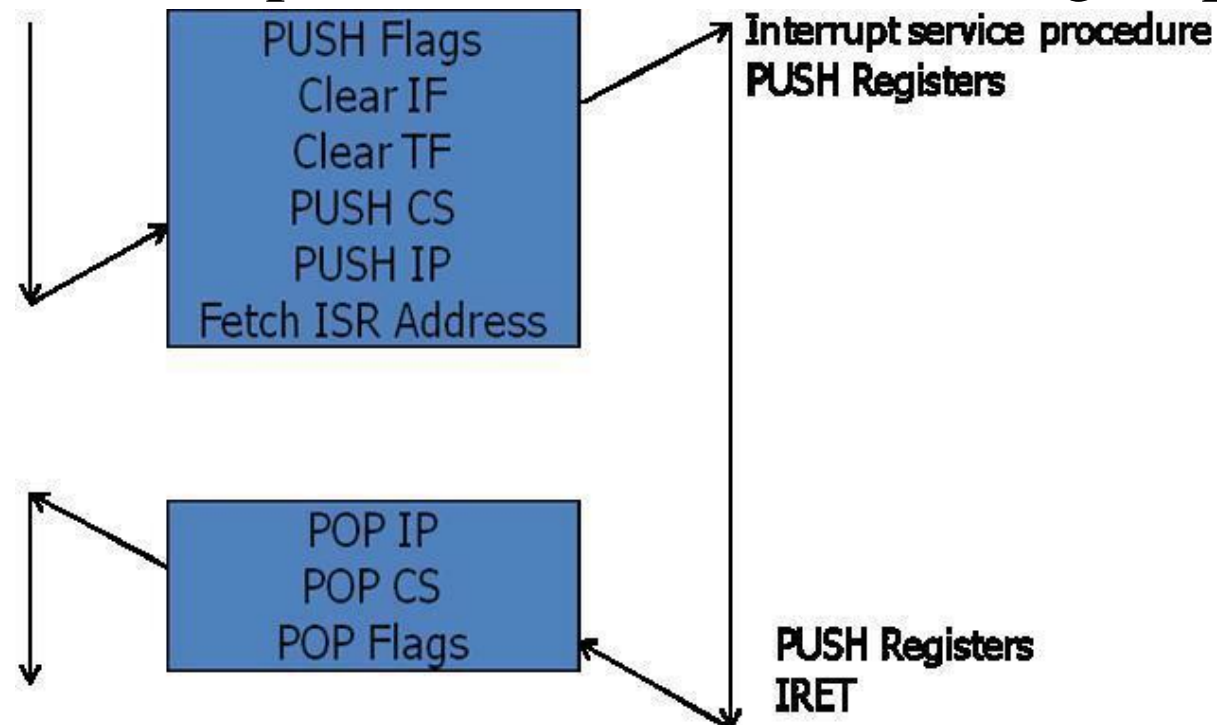
Process of interrupt Operation

- From the point of view of I/O unit
 - I/O device receives command from CPU
 - The I/O device then processes the operation
 - The I/O device signals an interrupt to the CPU over a control line.
 - The I/O device waits until the request from CPU.
- From the point of view of processor
 - The CPU issues command and then goes off to do its work.
 - When the interrupt from I/O device occurs, the processor saves its program counter & registers of the current program and processes the interrupt.
 - After completion for interrupt, processor requires its initial task.

Processor and Memory Interfacing:

Basic Interrupt Processing

- The occurrence of interrupt triggers a number of events, both in processor hardware and in software.
- The interrupt driven I/O operation takes the following steps.



Processor and Memory Interfacing:

- The I/O unit issues an interrupt signal to the processor for exchange of data between them.
- The processor finishes execution of the current instruction before responding to the interrupt.
- The processor sends an acknowledgement signal to the device that it issued the interrupt.
- The processor transfers its control to the requested routine called —Interrupt Service Routine (ISR) by saving the contents of program status word (PSW) and program counter (PC).

Processor and Memory Interfacing:

- The processor now loads the PC with the location of interrupt service routine and the fetches the instructions. The result is transferred to the interrupt handler program.
- When interrupt processing is completed, the saved register's value are retrieved from the stack and restored to the register.
- Finally it restores the PSW and PC values from the stack.

The figure summarizes these steps. The processor pushes the flag register on the stack, disables the INTR input and does essentially an indirect call to the interrupt service procedure.

An IRET function at the end of interrupt service procedure returns execution to the main program.

Processor and Memory Interfacing:

The Bus Arbitration:

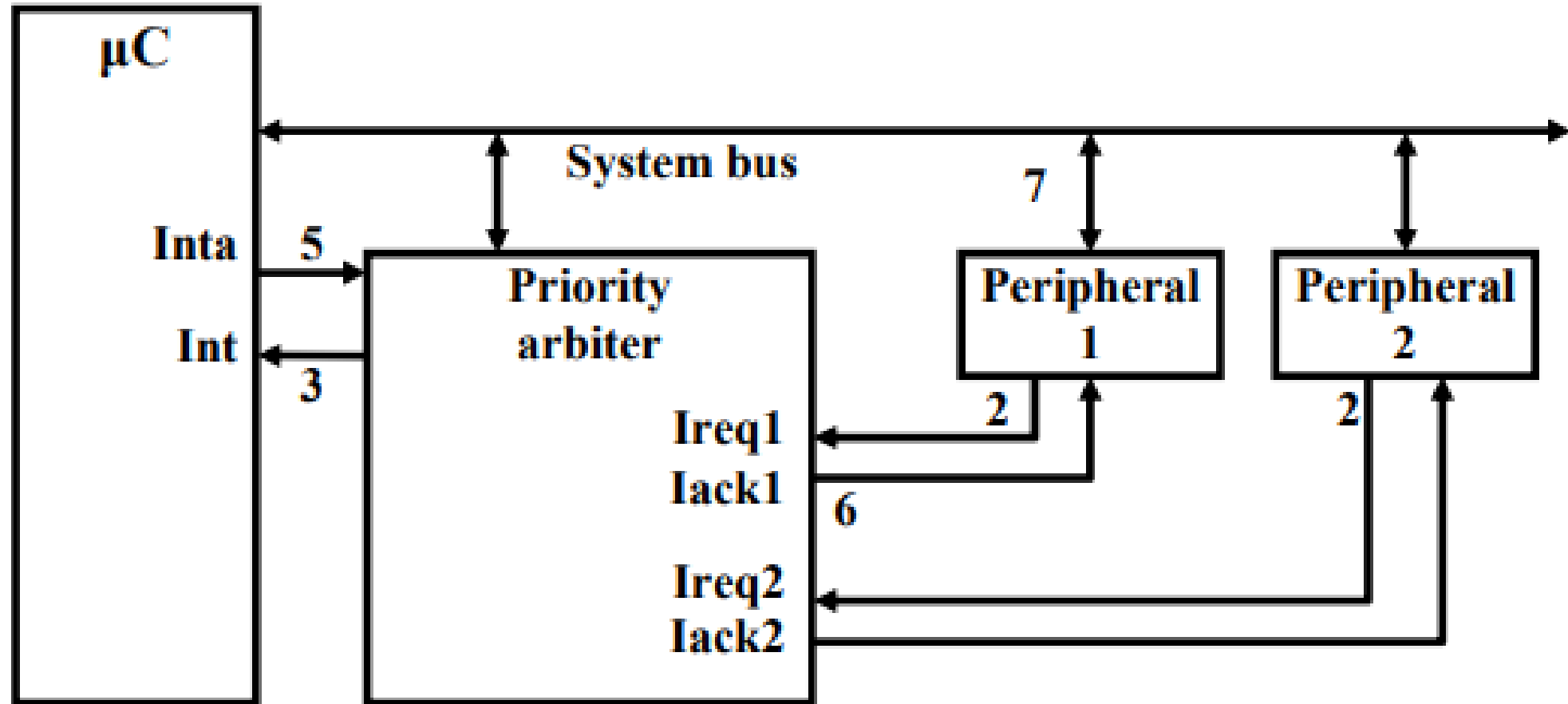
- The device that is allowed to initiate transfers on the bus at any given time is called the bus master.
- When the current bus master relinquishes its status as the bus master, another device can acquire this status.
- The process by which the next device to become the bus master is selected and bus mastership is transferred to it is called bus arbitration.
- When there are more than one device need interrupt service then they have to be connected in specific manner. The processor responds to each one of them. This is called **Arbitration**. The method can be divided into following.
 - Priority Arbiter
 - Daisy Chain Arbiter

Processor and Memory Interfacing:

Priority Arbiter

- Peripheral devices are connected with the processor and a priority arbiter is attached with processor selects the connection of peripheral with bus on the basis of fixed priority basis or round robin approach.
- If arbiter selects the PDs in fixed priority basis then such arbiter is called fixed priority arbiter.
- The round robin arbiter assigns a bus connection with PDs for a fixed interval of time and transfer to next connected device.
- The service taken by a device gets bus after a complete round. For example:

Processor and Memory Interfacing:



Processor and Memory Interfacing:

Let us assume that the Priority of the devices are Device1 > Device 2

.....then priority arbiter works as:

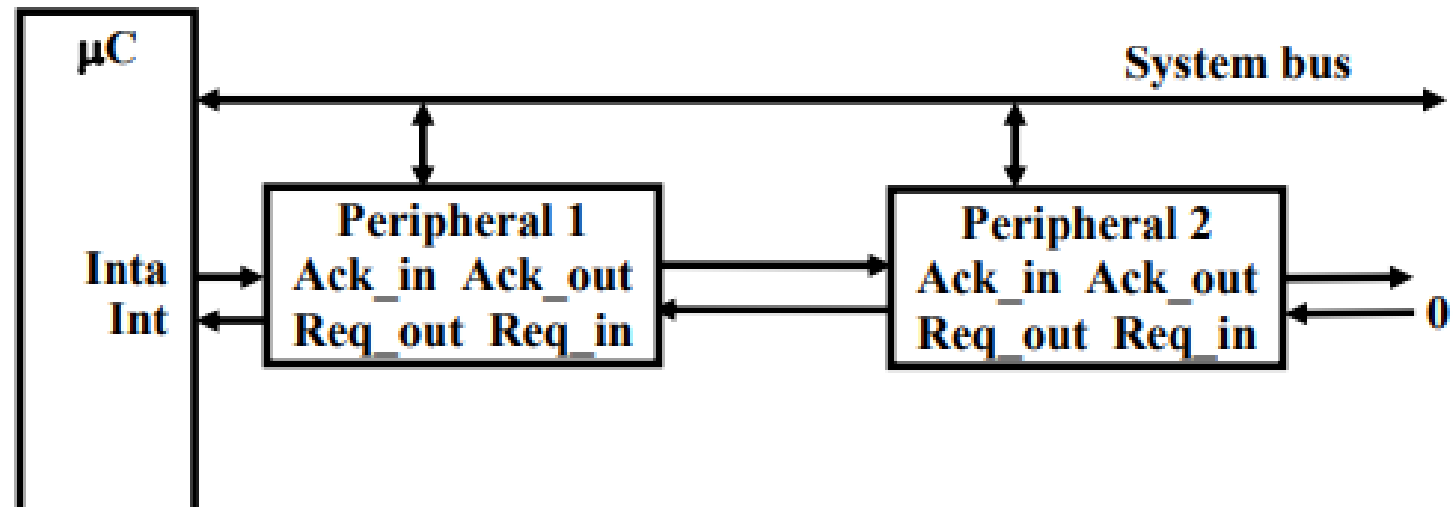
- The Processor is executing its program.
- Peripheral1 needs servicing so asserts **Ireq1**. Peripheral2 also needs servicing so asserts **Ireq2**.
- Priority arbiter sees at least one **Ireq** input asserted, so asserts **Int**.
- Processor stops executing its program and stores its state.
- Processor asserts **Inta**.
- Priority arbiter asserts **Iack1** to acknowledge Peripheral1.
- Peripheral1 puts its interrupt address vector on the system bus
- Processor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).

Processor and Memory Interfacing:

Thus in case of simultaneous interrupts the device with the highest priority will be served.

Daisy Chain Arbiter

In this case the peripherals needing interrupt service are connected in a chain as shown in Fig. The requests are chained and hence any device interrupting shall be transmitted to the CPU in a chain.



Processor and Memory Interfacing:

Let us assume that the Priority of the devices are Device1 > Device 2 ... then daisy chain arbiter works as:

- The Processor is executing its program.
- Any Peripheral needs servicing asserts **Req out**. This **Req out** goes to the **Req in** of the subsequent device in the chain
- Thus the peripheral nearest to the μC asserts Int.
- The processor stops executing its program and stores its state.
- Processor asserts **Inta** the nearest device.
- The **Inta** passes through the chain till it finds a flag which is set by the device which has generated the interrupt.

Processor and Memory Interfacing:

- The interrupting device sends the Interrupt Address Vector to the processor for its interrupt service subroutine.
- The processor jumps to the address of ISR read from data bus, ISR executes and returns.
- The flag is reset.

The processor now check for the next device which has interrupted simultaneously. In this case the device nearest to the processor has the highest priority.

Processor and Memory Interfacing:

Network-oriented arbitration

When multiple microprocessors share a bus (sometimes called a network) and Arbitration typically built into bus protocol. Separate processors may try to write simultaneously causing collisions such that

- Data must be resent
- Don't want to start sending again at same time

It is typically used for connecting multiple distant chips.

6.1 Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

- Sensors are essential components in many embedded systems as they allow the system to gather information about the environment. Depending on the sensor type, the data is either in **analog** or **digital** form. Let's discuss interfacing both **analog** and **digital** sensors with a microcontroller like Arduino for common sensors such as **temperature**, **humidity**, and **motion**.

1. Analog Sensors

- Analog sensors provide an **analog output** which is a continuous voltage (usually between 0 and 5V for most microcontrollers). The voltage is typically proportional to the measured physical quantity. For instance, a temperature sensor like the **LM35** provides an output voltage that is linearly proportional to the temperature.

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Example of Analog Sensors:

- **LM35 Temperature Sensor:** Provides a voltage of 10mV per °C.
- **LDR (Light Dependent Resistor):** Varies its resistance based on light intensity.
- **Flex Sensor:** Measures bending or flexing (changes resistance).
- **Potentiometer:** Provides a voltage based on the position of a knob.

Example Circuit for Interfacing an Analog Temperature Sensor (LM35) with Arduino:

- **LM35 Pinout:**
 - **VCC** → 5V
 - **GND** → Ground
 - **OUT** → Analog Pin (e.g., A0 on Arduino)

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Code to Read LM35 Temperature Sensor:

```
int tempPin = A0; // Pin connected to LM35 (analog input)
float voltage, temperature;
void setup() {
  Serial.begin(9600); // Start serial communication
}
void loop() {
  int sensorValue = analogRead(tempPin); // Read analog value from LM35
  voltage = sensorValue * (5.0 / 1023.0); // Convert the reading to voltage
  temperature = voltage * 100; // LM35 gives 10mV per degree Celsius
  Serial.print("Temperature: ");
```

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

```
Serial.print(temperature);  
Serial.println(" °C");  
delay(1000); // Wait for 1 second before taking the next reading }
```

Explanation of Code:

- The `analogRead()` function reads the analog value from pin A0, which is connected to the **OUT** pin of the LM35.
- The value is converted to voltage, and then the temperature is calculated based on the sensor's characteristics (10mV/°C).
- The temperature is displayed on the Serial Monitor.

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

2. Digital Sensors

- Digital sensors provide **binary output** (either HIGH or LOW, corresponding to 1 or 0) and are often easier to interface because they usually require only one data line. Some digital sensors, however, might use a communication protocol like **I2C** or **SPI**.

Example of Digital Sensors:

- **DHT11/DHT22**: Temperature and Humidity Sensor (Digital)
- **PIR Motion Sensor**: Detects motion (Digital output)
- **Reed Switch**: Detects the opening/closing of a door (Digital output)

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Example Circuit for Interfacing DHT11 Temperature and Humidity Sensor:

- The **DHT11** sensor provides both temperature and humidity data through a single digital signal. You will need a pull-up resistor (usually $10k\Omega$) to ensure proper communication.
- **DHT11 Pinout:**
 - **VCC** → 5V
 - **GND** → Ground
 - **DATA** → Digital Pin (e.g., Pin 7 on Arduino)

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Code to Read DHT11 Temperature and Humidity:

```
#include <DHT.h>

// Define the sensor pin
#define DHTPIN 7

#define DHTTYPE DHT11 // DHT 11 or DHT 22

DHT dht(DHTPIN, DHTTYPE); // Create an instance of the DHT sensor

void setup()
{ Serial.begin(9600); // Start serial communication
  dht.begin(); // Initialize the DHT sensor
  1/8/2025 }
}
```


Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

```
void loop() {  
  // Reading temperature or humidity takes about 250 milliseconds!  
  float h = dht.readHumidity(); // Read humidity (in percentage)  
  float t = dht.readTemperature(); // Read temperature in Celsius  
  
  // Check if any reading failed and exit early  
  if (isnan(h) || isnan(t)) {  
    Serial.println("Failed to read from DHT sensor!");  
    return; }  
}
```

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

```
// Print the results
Serial.print("Humidity: ");
Serial.print(h);
Serial.print(" %\t");
Serial.print("Temperature: ");
Serial.print(t);
Serial.println(" *C");
delay(2000); // Wait for 2 seconds before next reading }
```

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Explanation of Code:

- The **DHT** library is used to communicate with the DHT11 sensor.
- The sensor is initialized in `setup()`, and temperature and humidity values are read and printed to the Serial Monitor in the `loop()`.

3. Motion Sensors (PIR Sensor)

- The **PIR (Passive Infrared) Motion Sensor** detects changes in infrared light and is commonly used for motion detection applications. It outputs a digital signal: **HIGH** when motion is detected, and **LOW** when there is no motion.

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Example Circuit for Interfacing a PIR Motion Sensor:

- **PIR Pinout:**
 - **VCC** → 5V, **GND** → Ground
 - **OUT** → Digital Pin (e.g., Pin 8 on Arduino)

Code to Detect Motion Using PIR Sensor:

- `int pirPin = 8; // Pin connected to PIR sensor (digital input)`
- `int pirState = LOW; // Initial state of the PIR sensor`
- `void setup() {`
- `pinMode(pirPin, INPUT); // Set PIR pin as input`

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

```
Serial.begin(9600); // Start serial communication
} void loop() {
  pirState = digitalRead(pirPin); // Read the state of the PIR sensor
  if (pirState == HIGH) { // Motion detected
    Serial.println("Motion Detected!");
  } else {
    Serial.println("No Motion");
  } delay(500); // Wait for half a second
}
```

Sensor Interfacing: Analog and Digital Sensors (e.g., Temperature, Humidity, Motion)

Explanation of Code:

- The PIR sensor output is connected to a digital input pin (Pin 8).
- The `digitalRead()` function checks the PIR sensor's output. If the sensor detects motion, the state will be HIGH and the message "Motion Detected!" will appear in the Serial Monitor. Otherwise, "No Motion" will be printed.

6.2 Actuator interfacing: Motor Control DC

- DC Motors are very simple rotary actuators that transform electrical energy into a mechanical rotation at a specific torque.
- We typically use DC motors for applications that require moving objects (e.g. Robotics, Automation, UAV/UGV, etc).

Controlling a DC motor includes:

- (1) controlling the motor's speed, and
- (2) controlling the motor's direction of rotation (CW or CCW).
- Here, you'll learn how to control a DC motor's speed & direction using Arduino and L298N motor driver IC.

Actuator interfacing: Motor Control DC

A typical DC motor will have the following characteristics:

- Torque (in kg.cm)
- Rated Rotation Speed (RPM)
- Rated Full-Load current (e.g. 2A)
- Rated No-Load current (e.g. 0.2A)
- Rated voltage for operation (e.g. 12v)

Actuator interfacing: Motor Control DC

1- DC Motor Direction Control

- When electrical current passes through the motor's winding (coils) that are arranged within a fixed magnetic field (Stator).
- The current generates a magnetic field in the coils. This in turn causes the coil assembly (Rotor) to rotate, as each coil is pushed away from the like-pole and attracted to the unlike-pole of the stator.
- Reversing the direction of current flow in the coil translates to an inversion in the direction of the rotor's magnetic field. Which in turn applies an inversed torque to each side of the coil resulting in a reverse direction in the rotation.

Actuator interfacing: Motor Control DC

we'll be using a 2-channel H-Bridge motor driver called “**L298N**” which can be used to control the direction and speed of up to 2 DC motors at the same time.

DC Motor Speed Control

Controlling the speed of a DC motor is basically achieved in a couple of ways as shown below:

- Variable Supply Voltage
- Using PWM-Controlled Gate

Actuator interfacing: Motor Control DC

1- **Variable Supply Voltage:** can be achieved using voltage regulation circuitries. However, there are too many limitations and drawbacks to such a method that make it less practical to pursue. But theoretically, it just works and does the job. Varying the supply voltage will definitely control the motor's speed accordingly.

2- **Using a PWM-Controlled Gate:** This is the most common technique for motor control applications. It's basically done by isolating one of the power source rails from the H-Bridge circuitry using a transistor. Hence, creating an open circuit with the (Ground or V_M^+).

Actuator interfacing: Motor Control DC

- We use Arduino PWM output to control the speed of a DC motor. By varying the duty cycle for the motor output enable transistor, we can achieve motor speed control .
- Arduino boards have several PWM output pins usually. Those pins are designated with a (~) mark next to the pin number on the board.
- **Pulse Width Modulation (PWM)** is a technique for generating a continuous HIGH/LOW alternating digital signal and programmatically controlling its pulse width and frequency. Certain loads like (LEDs, DC Motors, etc) will respond to the **average voltage** of the signal which gets higher as the PWM signal's pulse width is increased.

Actuator interfacing: Motor Control DC

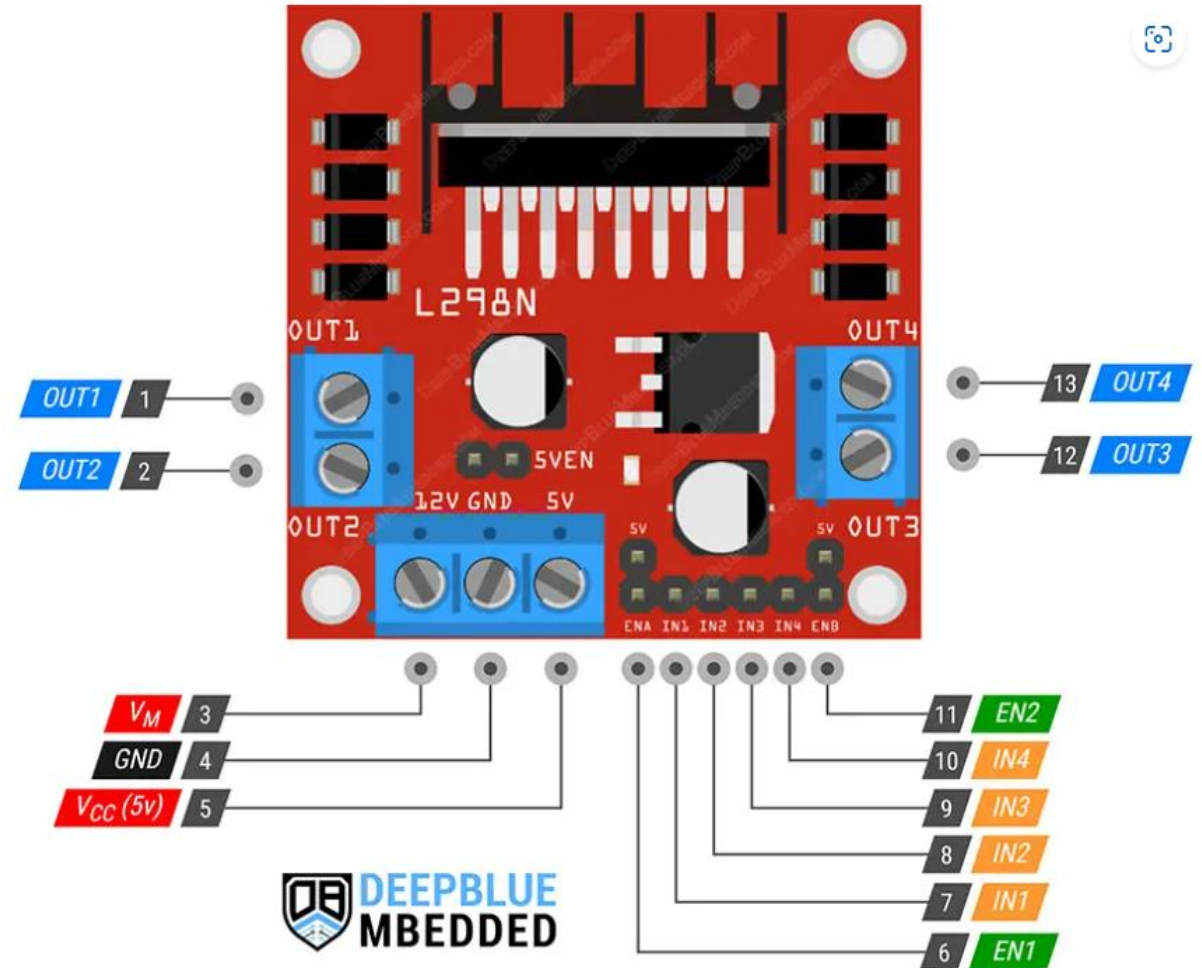
- The LED gets brighter as the pulse width (duty cycle) increases, and it gets dimmer as the pulse width decreases.
- And this is typically what also happens with a DC motor's speed, it gets faster or slower as the PWM's duty cycle changes.

L298N Motor Driver

- The L298N is a dual-channel H-Bridge motor driver IC capable of driving two DC motors or a single bipolar stepper motor. The ability to drive two separate DC motors makes it an ideal solution for simple two-wheel robotic vehicles.

Actuator interfacing: Motor Control DC

Here is the truth table for direction control input pins and what action the driver takes in each binary state combination for those input pins according to the L298N IC's datasheet.



Actuator interfacing: Motor Control DC

IN1	IN2	EN1	Action
0	1	1	Turn Forward
1	0	1	Turn Backward
0	0	1	Motor Brake (Fast Stop)
1	1	1	Motor Brake (Fast Stop)
X	X	0	Hi-Z (Motor Free-Run Stop)

Actuator interfacing: Motor Control DC

```
/* * LAB Name: Arduino DC Motor Control With L298N Driver
```

```
#define BTN_PIN 4    //4 IO pins
```

```
#define IN1_PIN 8
```

```
#define IN2_PIN 7
```

```
#define EN1_PIN 11
```

```
unsigned long T1 = 0, T2 = 0;
```

```
uint8_t TimeInterval = 5; // 5ms
```


Actuator interfacing: Motor Control DC

```
void setup() {  
  pinMode(BTN_PIN, INPUT); // we'll set the pinMode for the IO pins  
  pinMode(EN1_PIN, OUTPUT);  
  pinMode(IN1_PIN, OUTPUT); // initialize the motor direction control output  
  pins (IN1 & IN2).  
  pinMode(IN2_PIN, OUTPUT);  
  digitalWrite(IN1_PIN, HIGH);  
  digitalWrite(IN2_PIN, LOW);  
}
```

Actuator interfacing: Motor Control DC

```
void loop() { //Check the timer-based millis function to execute the core logic every 5ms.
```

```
    T2 = millis();
```

```
    if( (T2-T1) >= TimeInterval) // Every 5ms
```

```
    { // Read The Direction Control Button State
```

```
        if (debounce()) {
```

```
            digitalWrite(IN1_PIN, !digitalRead(IN1_PIN));
```

```
            digitalWrite(IN2_PIN, !digitalRead(IN2_PIN));
```

```
        } // Read The Potentiometer & Control The Motor Speed (PWM)
```

```
        analogWrite(EN1_PIN, (analogRead(A0)>>2));
```

```
        T1 = millis();    } }
```

Actuator interfacing: Motor Control DC

```
bool debounce(void) //This function is used to debounce the push button input pin
{
    static uint16_t btnState = 0;
    btnState = (btnState<<1) | (!digitalRead(BTN_PIN));
    return (btnState == 0xFFFF);
}
```

Actuator interfacing: PWM for Controlling Brightness

PWM (Pulse Width Modulation) is a technique used to control the power delivered to devices such as motors, LEDs, and other actuators. It works by varying the duty cycle of a square wave signal, which adjusts the average power delivered to the load. This is especially useful for controlling the brightness of LEDs, the speed of motors, and other actuator behaviors.

Key Components of PWM:

- **Duty Cycle:** The percentage of time the signal is HIGH in one complete cycle. A higher duty cycle means more power delivered.
- **Frequency:** The number of complete cycles of the PWM signal per second. Typically, this is constant, but the duty cycle changes.

Actuator interfacing: PWM for Controlling Brightness

For example:

- **0% Duty Cycle:** The signal is always off.
- **50% Duty Cycle:** The signal is on for half the time and off for the other half.
- **100% Duty Cycle:** The signal is always on.

Common Uses of PWM:

- **LED Brightness Control:** By varying the duty cycle of PWM, you can adjust the brightness of an LED.
- **Motor Speed Control:** PWM is used to control the speed of DC motors by adjusting the average voltage supplied to the motor.
- **Servo Motor Control:** Servos use PWM for positioning.

Actuator interfacing: PWM for Controlling Brightness

1. PWM for Controlling LED Brightness

- PWM is an excellent way to control the brightness of an LED. By varying the duty cycle of the PWM signal, you can simulate analog voltage levels and adjust the perceived brightness of the LED.

Example Circuit for Controlling LED Brightness:

- **LED** → Connected to a digital PWM-capable pin (e.g., Pin 9 on Arduino).
- **Current-limiting Resistor** → Typically 220Ω to 330Ω .

Actuator interfacing: PWM for Controlling Brightness

Arduino Code Example to Control LED Brightness:

```
int ledPin = 9; // Pin connected to the LED
int brightness = 0; // Initial brightness
int fadeAmount = 5; // Amount of change in brightness

void setup() {
  pinMode(ledPin, OUTPUT); // Set LED pin as an output
}

void loop() {
  // Adjust brightness by changing the duty cycle of the PWM signal
  analogWrite(ledPin, brightness);
```

Actuator interfacing: PWM for Controlling Brightness

```
// Change the brightness for the next loop iteration
```

```
brightness = brightness + fadeAmount;
```

```
// Reverse the direction of the fade when the brightness reaches the limits
```

```
if (brightness <= 0 || brightness >= 255) {
```

```
fadeAmount = -fadeAmount;
```

```
}
```

```
delay(30); // Wait for a short period to smooth the fade
```

```
}
```


Actuator interfacing: PWM for Controlling Brightness

In this example:

- The `analogWrite()` function is used to generate a PWM signal on `ledPin` with a duty cycle between 0 (off) and 255 (full brightness).
- The brightness gradually fades between low and high values.

2. PWM for Controlling Motor Speed

- PWM can also be used to control the speed of DC motors. By adjusting the duty cycle, you control how much voltage is delivered to the motor, and thus how fast it spins.

Example Circuit for Controlling DC Motor Speed:

- **DC Motor** → Connected to a motor driver (such as L298N or L293D) or directly to an H-Bridge.
- **PWM Pin** → Connected to the motor driver's control input.

Actuator interfacing: PWM for Controlling Speed

Arduino Code Example to Control DC Motor Speed:

- `int motorPin = 9; // Pin connected to the motor driver (PWM pin)`
 - `int motorSpeed = 0; // Speed of the motor (0 to 255)`
 - `void setup() {`
 - `pinMode(motorPin, OUTPUT); // Set motor pin as an output`
 - `}`
 - `void loop() {`
- `// Gradually increase the motor speed from 0 to 255 (full speed)`
- `for (motorSpeed = 0; motorSpeed <= 255; motorSpeed++) {`
- `analogWrite(motorPin, motorSpeed);`

Actuator interfacing: PWM for Controlling Speed

```
delay(20); // Wait 20ms before the next increment
}
// Gradually decrease the motor speed from 255 to 0 (stop)
for (motorSpeed = 255; motorSpeed >= 0; motorSpeed--)
{
    analogWrite(motorPin, motorSpeed);
    delay(20); // Wait 20ms before the next decrement
}
}
```

Actuator interfacing: PWM for Controlling Speed

In this example:

- The `analogWrite()` function controls the speed of the motor by adjusting the duty cycle of the PWM signal.
- The motor speed increases from 0 (stop) to full speed (255), and then decreases back to 0 (stop).

6.3 Display Interfacing

1. Interfacing an LCD with Arduino

- Most commonly used LCDs are 16x2 (16 characters by 2 rows), and they use the **HD44780** controller. Here's how to interface it:

Required Components:

- 16x2 LCD (HD44780 controller)
- Arduino (e.g., Uno, Nano, Mega)
- 10k potentiometer (for contrast adjustment)
- Jumper wires
- Breadboard

Display Interfacing

Pinout of 16x2 LCD (HD44780):

- **VSS** → GND
- **VDD** → 5V
- **V0** → Contrast (connect to the middle pin of the potentiometer)
- **RS** → Register Select (Pin 12 on Arduino)
- **RW** → Read/Write (GND for write mode)
- **E** → Enable Pin (Pin 11 on Arduino)
- **D4** → Data Pin 4 (Pin 5 on Arduino), **D5** → Data Pin 5 (Pin 4 on Arduino)
- **D6** → Data Pin 6 (Pin 3 on Arduino), **D7** → Data Pin 7 (Pin 2 on Arduino),
- **A** → Anode for backlight (5V)
- **K** → Cathode for backlight (GND)

Display Interfacing

```
#include <LiquidCrystal.h>

// Initialize the LCD (pin 12, 11, 5, 4, 3, 2 are connected to the LCD) LiquidCrystal
lcd(12, 11, 5, 4, 3, 2);

void setup() {
  // Start the LCD
  lcd.begin(16, 2); // 16x2 LCD
  // Print a message to the LCD
  lcd.setCursor(0, 0); // Set cursor to column 0, row 0
  lcd.print("Hello, World!");
  lcd.setCursor(0, 1); // Set cursor to column 0, row 1
  lcd.print("Arduino LCD"); }

void loop() { // Nothing to do here, the display will show the message }
```

Display Interfacing

2. Interfacing a Seven-Segment Display with Arduino

Seven-segment displays can either be **common anode** or **common cathode**. Most often, you'll find **common cathode** displays, where each segment of the display is connected to a ground (GND), and the common pin is connected to a voltage source.

Required Components:

- Common Cathode 7-segment display
- Arduino
- Resistors (220Ω to 330Ω for each segment)
- Jumper wires
- Breadboard

Display Interfacing

- **Pinout of a Common Cathode 7-Segment Display:**
- A 7-segment display typically has 7 segments labeled from **A to G** and a common cathode pin. The segment pins (A to G) correspond to specific Arduino pins.

Circuit Connections (assuming Common Cathode 7-segment):

- **Segment A** → Arduino Pin 2 (with 220Ω resistor)
- **Segment B** → Arduino Pin 3 (with 220Ω resistor)
- **Segment C** → Arduino Pin 4 (with 220Ω resistor)
- **Segment D** → Arduino Pin 5 (with 220Ω resistor)
- **Segment E** → Arduino Pin 6 (with 220Ω resistor)
- **Segment F** → Arduino Pin 7 (with 220Ω resistor)
- **Segment G** → Arduino Pin 8 (with 220Ω resistor)
- **Common Cathode Pin** → GND

Display Interfacing

- `int segPins[] = {2, 3, 4, 5, 6, 7, 8}; // Pins connected to segments A to G`
`// Digits in 7-segment display (bit patterns for common cathode)`

```
int digits[10][7] = {  
    {1, 1, 1, 1, 1, 1, 0}, // 0  
    {0, 1, 1, 0, 0, 0, 0}, // 1  
    {1, 1, 0, 1, 1, 0, 1}, // 2  
    {1, 1, 1, 1, 0, 0, 1}, // 3  
    {0, 1, 1, 1, 0, 1, 1}, // 4  
    {1, 0, 1, 1, 0, 1, 1}, // 5  
    {1, 0, 1, 1, 1, 1, 1}, // 6  
    {1, 1, 1, 0, 0, 0, 0}, // 7
```

Display Interfacing

```
{1, 1, 1, 1, 1, 1, 1}, // 8
{1, 1, 1, 1, 0, 1, 1} // 9
};

void setup() {
    // Set all segment pins as output
    for (int i = 0; i < 7; i++) {
        pinMode(segPins[i], OUTPUT);
    }

    // Display the digits from 0 to 9
    for (int num = 0; num < 10; num++) {
```

Display Interfacing

```
displayDigit(num);  
delay(1000); // Wait for 1 second  
}  
  
void loop() {  
  // Nothing to do here  
}  
  
void displayDigit(int num) {  
  for (int i = 0; i < 7; i++) {  
    digitalWrite(segPins[i], digits[num][i]);  
  }  
}
```

Display Interfacing

Key Points:

- **LCD:**

- Uses a controller (typically HD44780) to manage communication.
- Requires multiple control lines (RS, RW, E) and data lines (D4–D7 for 4-bit mode).
- Use a potentiometer to adjust the display contrast.

- **Seven-Segment Display:**

- Each segment can be individually controlled to form numbers.
- Requires current-limiting resistors (typically 220Ω to 330Ω).
- Can be controlled with simple digital outputs from the Arduino.

Display Interfacing

Advanced Notes:

- **Multiplexing (for multiple 7-segment displays):** When controlling more than one 7-segment display, multiplexing is commonly used to quickly switch between displays to create the illusion of them being displayed simultaneously.
- **LCD Libraries:** Arduino has the LiquidCrystal library, which simplifies the communication process with the LCD.