Politecnico di Torino

Project Title: **Digital Voltmeter with Embedded and FPGA Systems**

Submitted by: **Mohsen Zarei** (S343388)

Course: **Electronics for Embedded Systems**

Instructor: **Prof. Claudio Passerone**

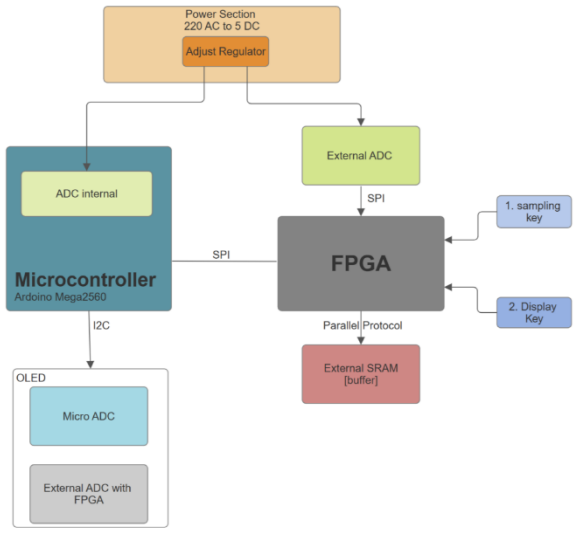
Institution: **Politecnico di Torino**

Date: **December 2024**

*** Table of Contents ***

Project Introduction

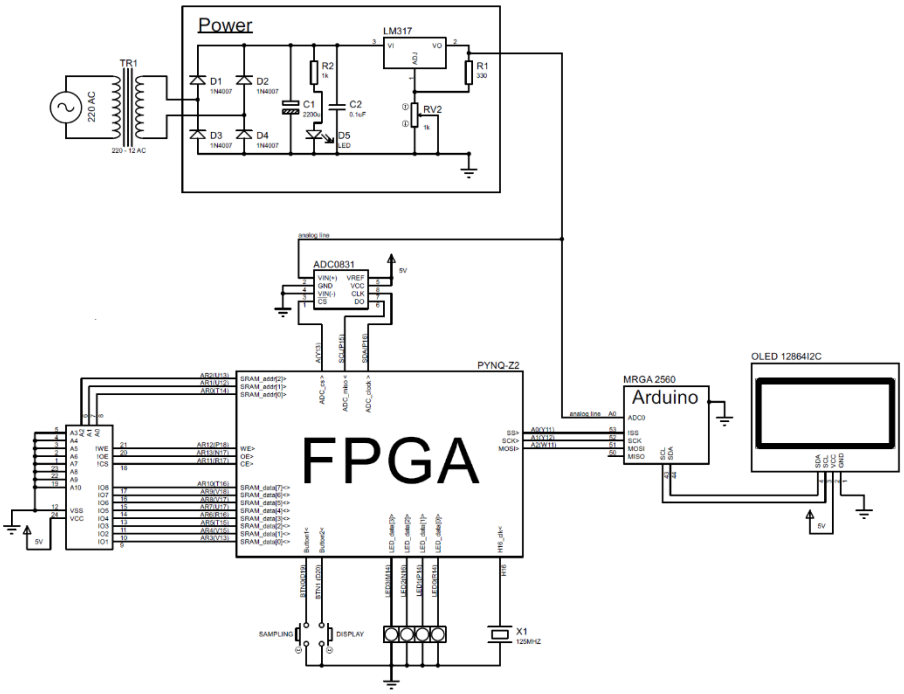
This project presents the design and implementation of a **digital voltmeter system** utilizing both an **embedded microcontroller (Arduino Mega 2560) and an FPGA** to perform **analog-to-digital conversion (ADC)** and voltage monitoring. The primary objective is to **compare the performance and precision of an internal microcontroller ADC with an external FPGA-based ADC**, while also integrating an **SRAM memory module (HY6116ALP)** for temporary data storage. The system samples voltage from a **linear power supply**, which converts a **220V AC input into an adjustable DC output**, and displays the results on an **OLED screen via I2C protocol**.



"Block Diagram of Components and Connections"

The FPGA is responsible for **interfacing with an external 8-bit ADC (ADC0831N) via SPI**, storing sampled voltage values in SRAM, and communicating with the microcontroller. The **Arduino Mega 2560**, acting as the primary processing unit, is configured at the **register level** for SPI and ADC operations. It samples voltage using its **10-bit internal ADC**, scales the values to match the external ADC's resolution for **comparison**, and displays the data on an **OLED screen**. The system operates via **two push buttons on the FPGA**, one for **initiating the sampling process**, and another for **transferring stored data to the microcontroller** for display.

This project integrates essential **embedded system concepts**, including **register-level programming, inter-device communication protocols (SPI & I2C), memory management, and power system design**. By demonstrating the differences in **precision, response time, and storage efficiency** between the two ADC implementations, the project provides valuable insights into real-world applications of **FPGA and microcontroller-based measurement systems**.



1. Linear Power Supply Introduction

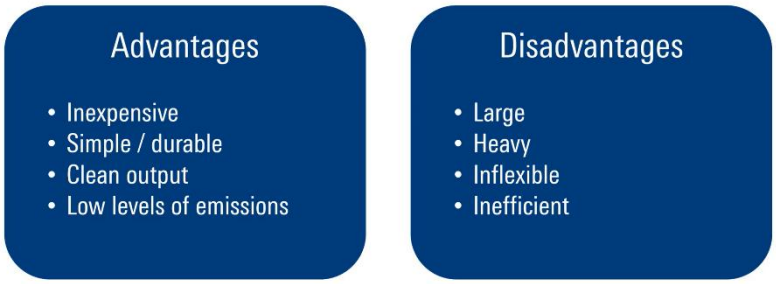
A power supply is a critical device that converts high-voltage mains power, delivered as alternating current (AC), into the low-voltage direct current (DC) required by most electronic devices. Mains power typically ranges from 100 to 240 volts AC, while electronic devices operate internally on low-voltage DC, often just a few volts.

In some cases, batteries provide this low-voltage DC, but they are not always practical and themselves rely on DC for recharging. A power supply ensures a steady, reliable DC output, transforming an electrical sine wave into a straight-line voltage.

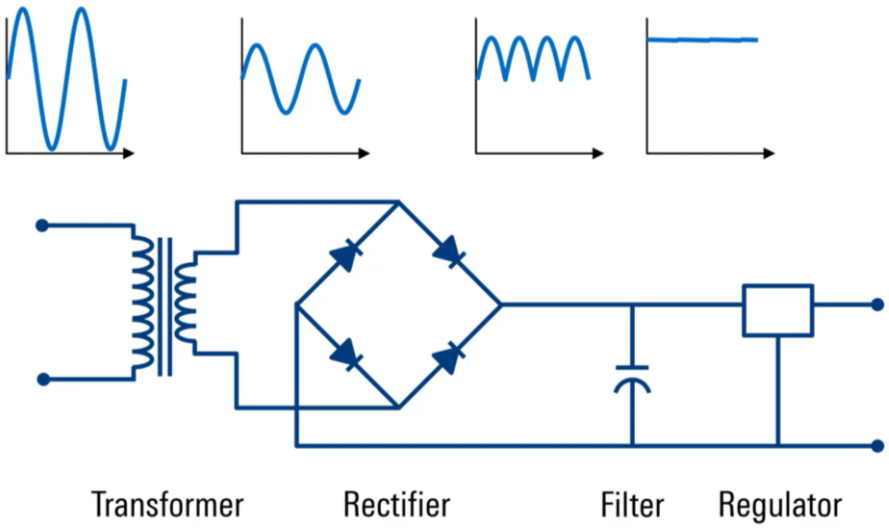
Power supplies are broadly categorized into two types:

- **Linear Power Supplies**: Simple, low-noise, and precise.
- **Switching Mode Power Supplies**: More complex but highly efficient.

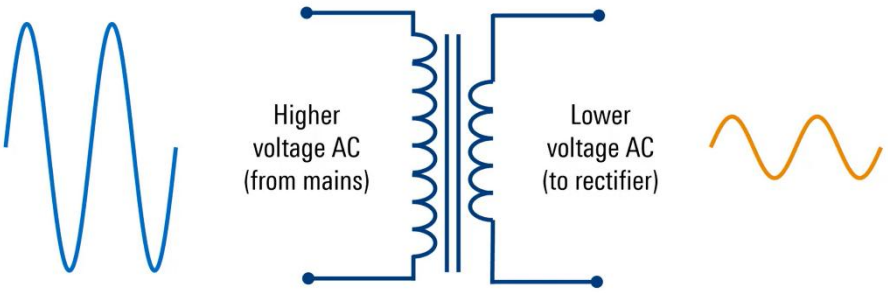
While linear power supplies have their advantages, such as simplicity and low noise, and disadvantages like lower efficiency, I chose this type for its stability and suitability for the applications such as ADC-based measurements.



A **linear power supply** converts high-voltage AC mains into low-voltage DC through four key stages:

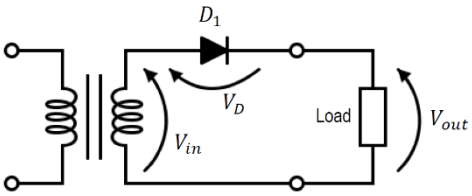


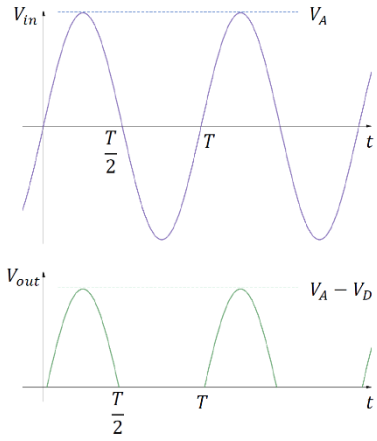
1. **Transformer**: Reduces high-voltage AC to a lower, manageable AC voltage. The transformer steps down high-voltage AC mains (100–240V) to a lower AC voltage suitable for further processing. For example, a 10:1 turns ratio reduces 240V AC to 24V AC. While this low-voltage AC can power devices like lamps and heaters, it is unsuitable for electronic circuits without rectification and smoothing to produce stable DC.



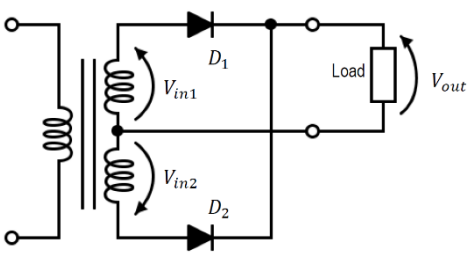
2. **Rectifier**: Converts the lower AC voltage into pulsating DC using diodes in a bridge configuration. After stepping down the AC voltage, the next stage is **rectification**, where diodes are used to convert AC into pulsating DC. Diodes allow current to pass in only one direction, enabling this transformation.

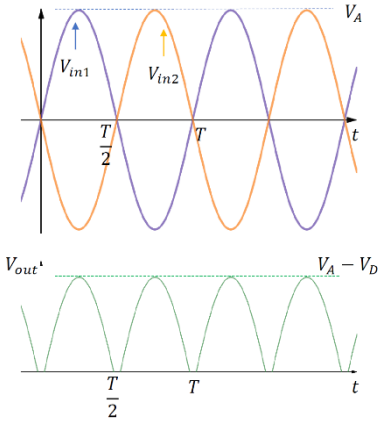
Half-Wave Rectifier: Uses one diode to convert only the positive half of the AC waveform; simple but inefficient.



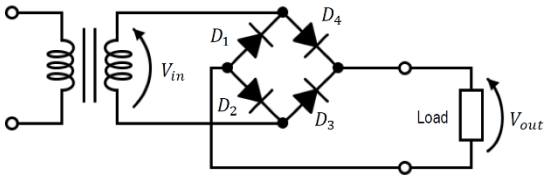


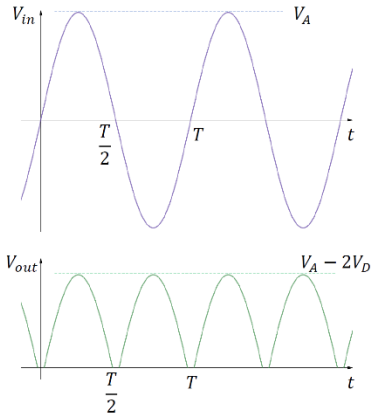
Full-Wave Rectifier: Uses two diodes and a center-tapped transformer to convert both halves of the AC waveform; more efficient.





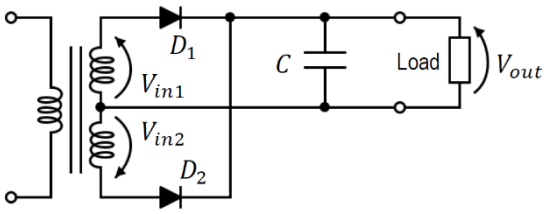
Bridge Rectifier: Uses four diodes to rectify both halves of the AC waveform without needing a center-tapped transformer; cost-effective and widely used.

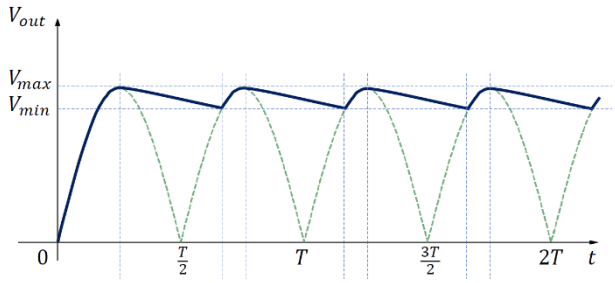




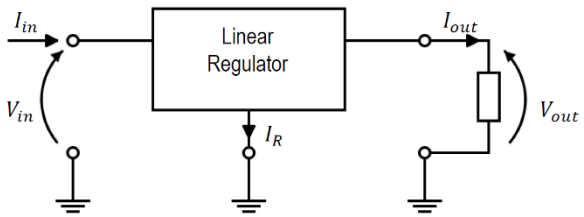
The most common implementation in linear power supplies is a **bridge rectifier**, which uses four diodes to convert the full AC sine wave into pulsating DC. This approach is cost-effective compared to using a center-tapped transformer with only two diodes.

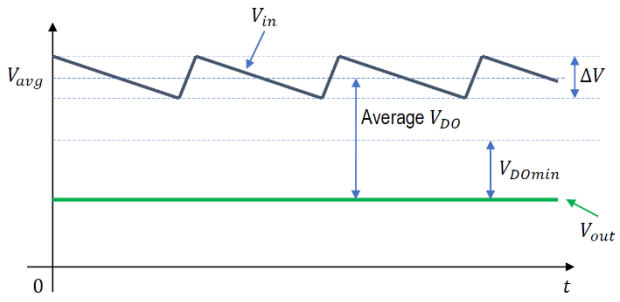
3. **Filtering**: Smooths the pulsating DC with capacitors to reduce ripples and provide a stable voltage. This is typically achieved using one or more capacitors, sometimes paired with inductors, to stabilize the voltage. Capacitors charge as the voltage rises and discharge as it falls, filling the gaps between waveform peaks. Larger capacitors provide better smoothing by maintaining voltage for longer periods, though some residual ripple may still remain. A simple filter consists of a single capacitor placed in parallel with the load, but more complex designs can include additional components for enhanced performance.





4. **Voltage Regulation**: After the filtering stage, the output may still have a residual ripple or voltage fluctuations, which depend on the filter capacitor's size. While increasing capacitance reduces ripple, it can make the design bulkier and more expensive. A voltage regulator addresses these issues by maintaining a constant output voltage, even with changing input conditions or load demands. It typically operates as a feedback loop, using a transistor and an op-amp to adjust the voltage dynamically. This regulation creates a stable DC output, satisfying the requirements of sensitive electronic components like ADCs.

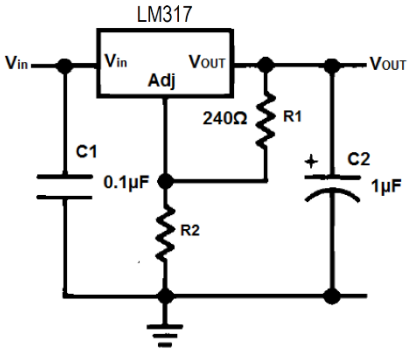




There are various voltage regulators available, such as the 78xx series (fixed positive output), 79xx series (fixed negative output), and LM337 (adjustable negative output). However, we use the **LM317** because it provides a variable output voltage, easily adjustable using external resistors or a potentiometer, making it suitable for applications requiring precise and adjustable power.

The output voltage of the LM317 is determined by the formula:

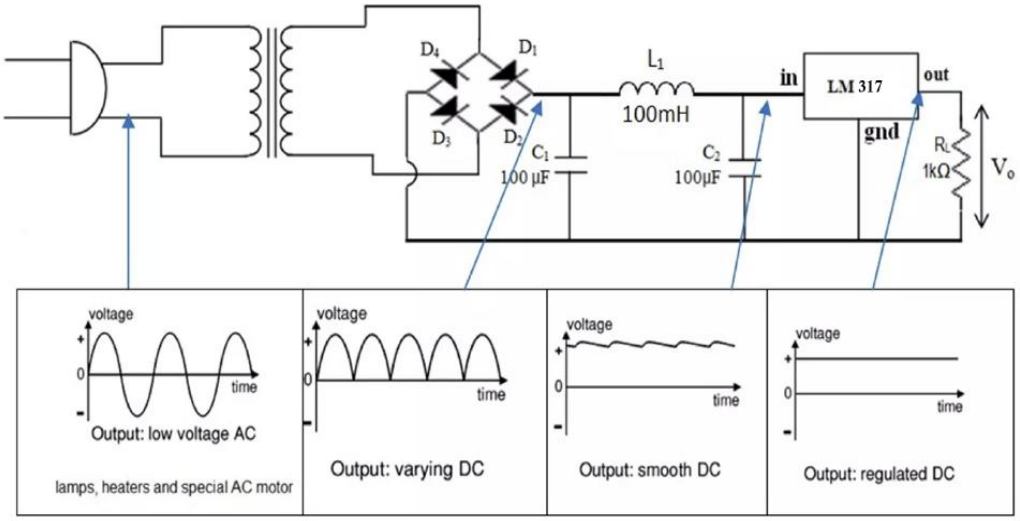
$$V_{out} = 1.25\left(1 + \frac{R_2}{R_1}\right)$$



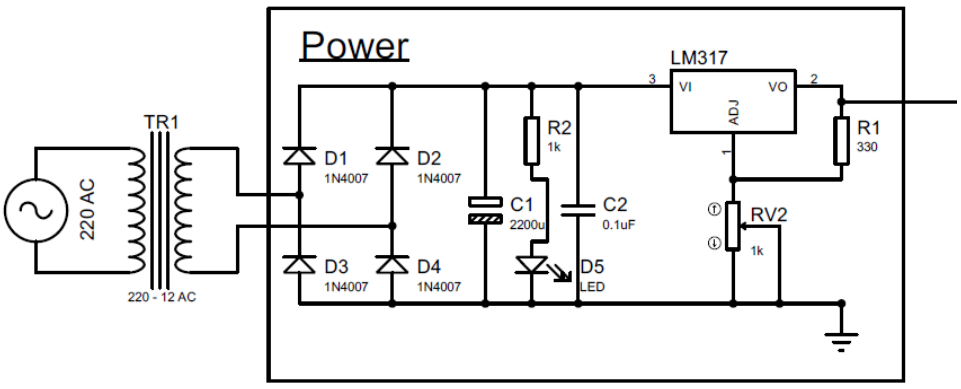
R2: Can be replaced with a potentiometer for variable output.

Capacitors: Recommended at input and output for improved stability and ripple rejection.

In summary, a linear power supply first steps down high-voltage mains AC to a lower voltage AC using a transformer. This lower AC voltage is then rectified to produce a pulsating positive DC voltage. A capacitor-based filter smooths the pulsating voltage into a mostly flat DC output. Optionally, a voltage regulator can be added to maintain constant output voltage despite variations in input voltage or load impedance.



Power Section on this Project:

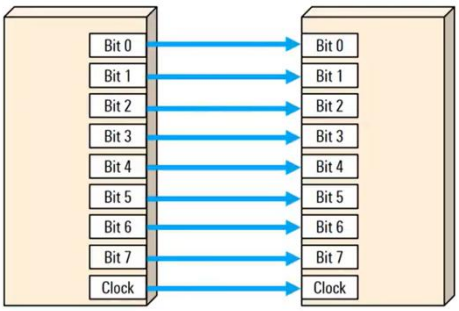


2. Interconnections

Interconnections are fundamental in embedded systems and electronic devices, facilitating the transfer of information (bits) between components such as microcontrollers, sensors, displays, and external devices like USB microphones and computers at various levels. The goal of interconnections is to ensure data is exchanged accurately, reliably, and efficiently while considering performance metrics like power consumption and cost. There are many ways of moving bits, but the different methods of transferring bits can be separated into two main categories:

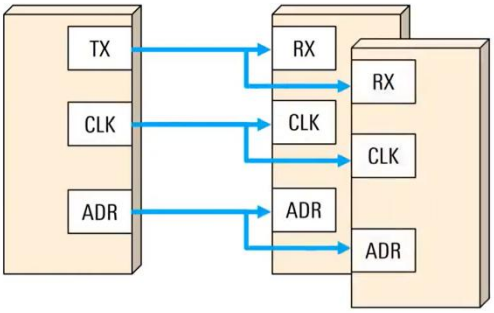
1. Parallel Interconnections:

- Transmit multiple bits simultaneously across several wires.
- Suitable for short distances and high-speed data transfer.
- Require more physical connections, making them expensive, bulky, and prone to noise over long distances.
- Simple timing and easy to analyze, commonly used in legacy systems like parallel ports.

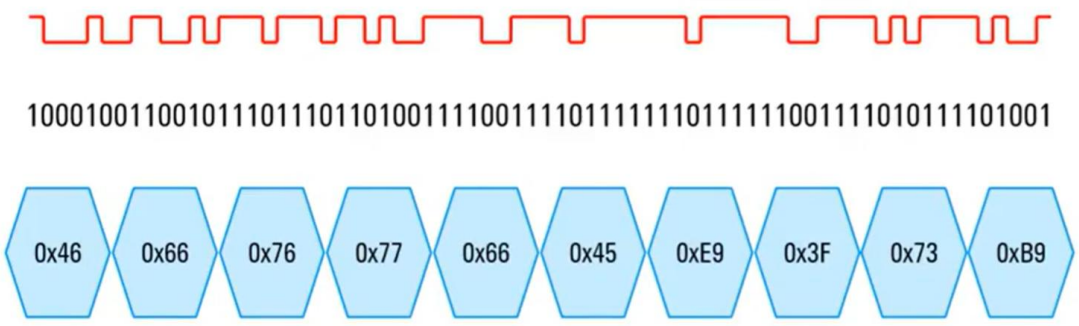


2. **Serial Interconnections**:

- Transmit bits sequentially over a single wire, often with additional for data, clock or control/addressing.
- Ideal for long-distance, cost-efficient communication and scenarios requiring minimal wiring.
- Generally, more complex and slower per cycle than parallel but scales better for higher throughput.

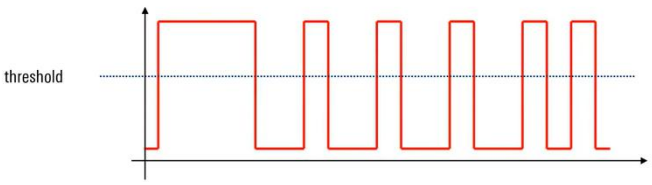


Serial communication is preferred over parallel for its efficiency in long-distance and multi-device applications, requiring fewer wires and being less prone to noise. Common protocols include **UART**, **I2C**, and **SPI**, as well as specialized protocols like **CAN**, **LIN**, and **FlexRay**, which are widely used in automotive systems and critical tasks.

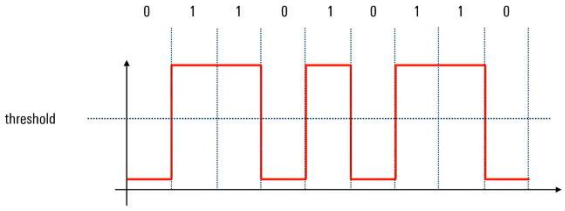


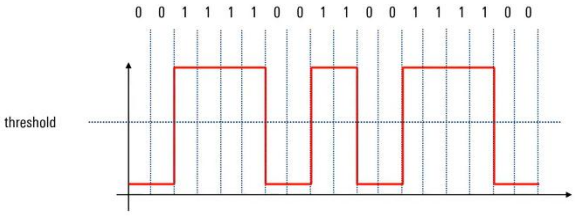
Although implementation details differ between protocols all serial protocols have four basic characteristics:

1. **Levels:** In serial protocols, **levels** refer to the way voltage is used to represent binary used, where the binary value is determined by the difference between two voltage lines, offering better noise immunity in challenging environments. Levels ensure the receiver can reliably decode the transmitted data.

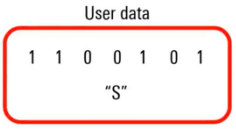


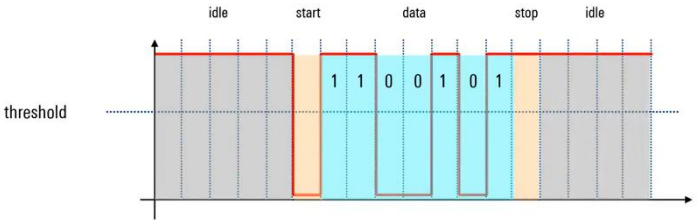
2. **Timing:** Simply defining voltage levels for 0 and 1 is not enough; the timing determines how fast bits are generated and distinguishes one bit from the next. Both the transmitter and receiver must use the same bit rate to decode data correctly. Proper timing is essential for reliable communication in serial protocols.



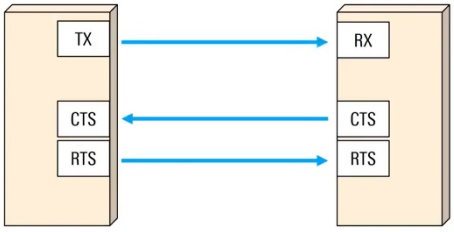


3. **Framing:** Frames define the structure of data transmission, specifying the purpose of each bit or group of bits. Typically frame includes a **start bit** to indicate the beginning, **data bits** to convey information, and a **stop bit** to mark the end. Understanding the frame structure is essential for decoding the transmitted data accurately. For instance, by knowing the frame structure we can extract the user data here the ASCII letter capital S from the serial bit stream as well as derive other information about the transmission.





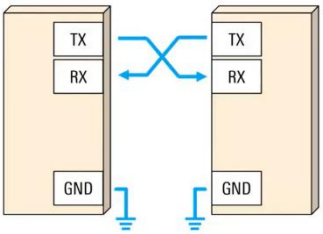
4. **Protocol:** is a set of rules that governs how information is encoded, transmitted, and exchanged between devices. It defines the types of messages exchanged, the circumstances under which they are sent, and how they are interpreted by the receiver.



Protocols can range from simple (e.g., sending data immediately without checking the receiver's readiness) to complex systems involving acknowledgments and error handling. Protocols ensure reliable and organized communication by establishing clear rules for how devices interact.
Among them, **UART**, **I2C**, and **SPI** are widely used and considered some of the most common protocols for communication between electronic devices.

2.1. UART Protocol

UART (Universal Asynchronous Receiver Transmitter) is a simple serial communication protocol that defines the rules for exchanging data between two devices. It uses only two main wires: **Transmit (TX)** and **Receive (RX)**, along with a ground connection. UART supports three communication modes: **simplex** (one-way), **half-duplex** (two-way but one at a time), and **full-duplex** (simultaneous two-way communication). Data is transmitted in the form of frames, which include specific structures to organize the information.

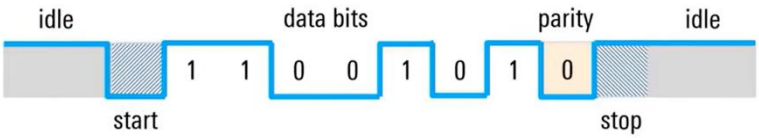


Due to its simplicity and efficiency in point-to-point communication, UART has been widely used in PCs, embedded systems, and devices like RS-232 interfaces and external modems. Although protocols like SPI, I^2C, Ethernet, and USB have emerged, UART is still widely used for low-speed, low-throughput applications due to its simplicity, low cost, and reliability.

One major advantage of UART is its asynchronous nature, eliminating the need for a shared clock, but requiring both transmitter and receiver to operate at the same prearranged baud rate and frame structure for proper communication.
in addition to having the same baud rate both sides of a your connection also have to use the same frame structure and parameters the best way to understand this is to look at a UART frame

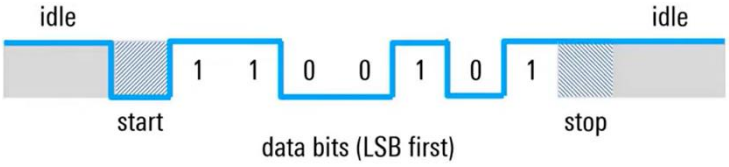
Common UART baud rates
4800
9600
19200
57600
115200

UART frames consist of **start bits**, **stop bits**, **data bits**, and an optional **parity bit**. High voltage levels represent logical ones (mark), while low levels represent logical zeros (space), but UART does not define specific voltage ranges for these levels. In the idle state, the line is held high, ensuring easy detection of a damaged line or transmitter when no data is being transmitted.



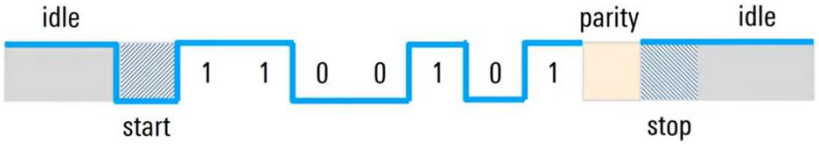
- **Start and Stop bits:**
  In UART communication, the **start bit** signals the beginning of data transmission by transitioning from the idle high state to a low state. Immediately following the start bit are the **data bits**, representing the user data. Once the data bits are transmitted, the **stop bit** marks the end of the frame by transitioning back to or remaining at the idle high state. An optional second stop bit can be added to provide additional time for the receiver to prepare for the next frame, though this is rarely used in practice.

- **Data bits:**
  The **data bits** in a UART frame carry the user data and follow immediately after the start bit. Typically, there are 5 to 9 data bits, with 7 or 8 bits being the most common. These bits are transmitted in **least significant bit (LSB) first** order. For example, to send the capital letter 'S' in 7-bit ASCII (binary 1010011), the bit sequence is reversed to 1100101 before transmission. After the data bits, the **stop bit** marks the end of the frame, returning the line to the idle state.



- **Parity bit:**
  A UART frame may include an optional parity bit for error detection, placed between the data bits and the stop bit. Its value depends on the parity type:
  - **Even Parity:** Ensures the total number of ones in the frame is even.
  - **Odd Parity:** Ensures the total number of ones is odd.

  For example, the ASCII code for 'S' (1010011) has four ones. With even, the parity bit is 0; with odd parity, it is 1. Parity bits can detect single-bit errors but not multiple-bit errors.

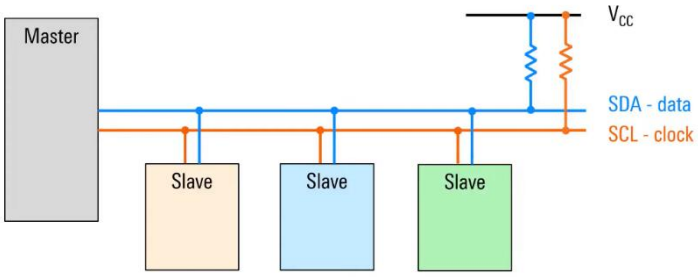


In summary, UART (Universal Asynchronous Receiver Transmitter) is a simple two-wire protocol for serial data exchange. Asynchronous communication eliminates the need for a shared clock, requiring both sides to operate at the same baud rate. Start and stop bits frame the data, and an optional parity bit provides basic error detection. While widely used for decades, UART has been partially replaced by faster technologies like SPI, I^2C, USB, and Ethernet in some applications.

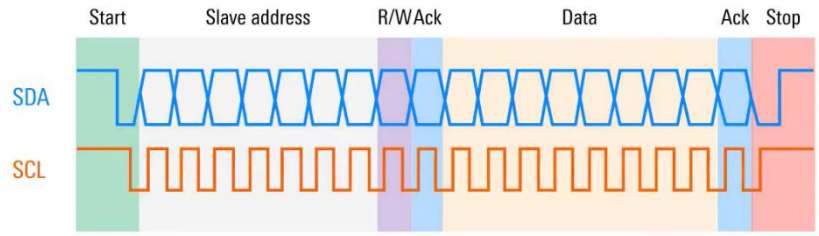
2.2. I2C Protocol

I2C (Inter-Integrated Circuit) is a synchronous serial protocol developed by Philips (now NXP) in 1982 for short-distance data communication. It uses a **master-slave architecture**, where both the master and slaves can send and receive data in a bi-directional, half-duplex mode.

I2C operates at different clock speeds with just two wires: **SDA (Serial Data)** and **SCL (Serial Clock)**, both connected to a voltage source (VCC or VDD) via pull-up resistors. Its flexible topology allows devices to be added or removed from the shared bus at any time, and data is exchanged in the form of frames, making it simple and widely used in embedded systems.

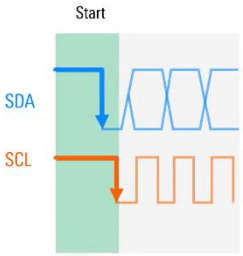


Frame: An I2C frame begins with the master initiating a **start condition**, then the **first byte** transmitted contains the **slave address** and the **read/write bit**, specifying the target slave and the operation type. The slave responds with an **acknowledgment**, after which data is transmitted and acknowledged byte by byte. The communication concludes with a **stop condition**, releasing the bus.



Start condition:

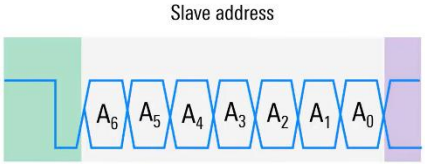
In I2C, both SDA and SCL lines are idle in the high state. The start condition occurs when a node pulls SDA low, followed by SCL low, claiming the bus and becoming the master. This prevents other nodes from taking control, reducing the risk of contention. Once the bus is claimed, the master starts sending the clock signal for communication.

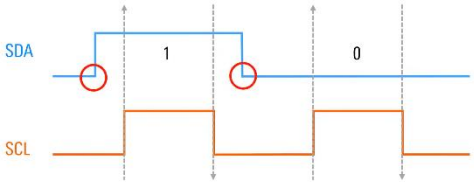


Slave Address:

After the start condition, the master sends the **slave address** to specify the target node for communication. Each node on the I2C bus must have a unique 7-bit address, transmitted with the most significant bit (MSB) first.

Although I2C supports 10-bit addresses, these are rarely used. Addresses can be hard-coded for each device or partially configured using external address lines or jumpers.





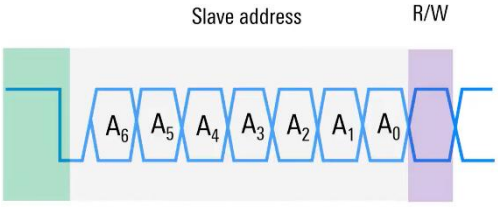
In I2C, **SDA** transitions only when **SCL** is low during data transmission. SDA remains stable between the rising and falling edges of the clock, with data being read in the middle of the clock pulse. An SDA transition while SCL is high signals a start or stop condition, making this timing relationship essential for proper communication.

Read/Write bit:

Following the slave address is the **read/write bit**, set by the master to indicate the operation type.

0 → means the master wants to write data to the slave,

1→ indicates a read operation.



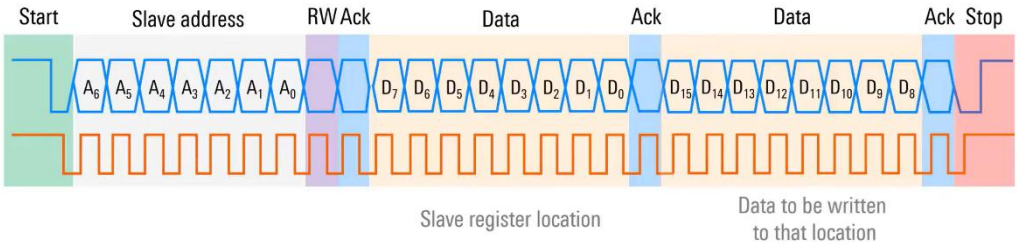
In standard I2C, where the slave address is 7 bits, the read/write bit is often treated as part of the address byte.

ACK Bit:

The **ACK bit** (Acknowledge) is sent by the receiver after each byte of data is received.
0 → indicates acknowledgment (ACK)
1 → indicates a negative acknowledgment (NACK).
Since the I2C bus is idle high, a NACK occurs if the receiver does not pull the line low.
An ACK following a **slave address** confirms the addressed slave is present and ready, while an ACK after a **data byte** confirms the data was received correctly.

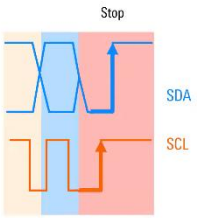
Data Byte(s):

After the address and ACK bit, the **data byte(s)** are transmitted, containing the actual information exchanged between the master and slave. This could be a memory address, register value, or other data. In I2C, data is always sent as 8-bit bytes with the **most significant bit (MSB)** first. Each transmitted byte is followed by an **ACK bit**, where the receiver pulls the line low (0) to confirm proper reception.

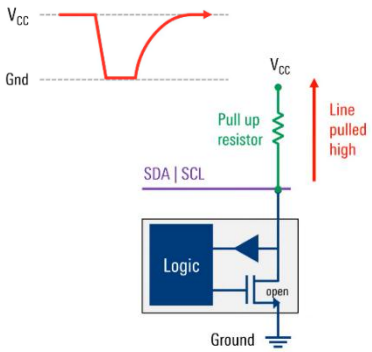


In I2C, multiple data bytes can be transmitted within a single frame by concatenating them. Each byte must be individually acknowledged by the receiver with an **ACK bit**. While I2C does not define the content or purpose of the data bytes, they often include instructions or values.
For instance, the first byte might specify a register address in the slave device, and the subsequent byte could contain the data to be written to that register. This flexibility makes I2C versatile for various applications.

Stop Condition:



The **stop condition** ends I2C communication. After the final data byte, **SCL** is released to high, followed by **SDA** transitioning to high. This transition on **SDA** while **SCL** is high signals the stop condition. The bus then becomes idle, ready for a new **start condition** to initiate the next communication.



I2C uses an **open-drain system**, where devices pull the shared lines (**SDA** and **SCL**) low but rely on **pull-up resistors** to return the lines to a high (idle) state. The value of the pull-up resistors impacts the bus speed and power consumption: higher resistance saves power but slows the return to high, while lower resistance allows faster speeds but uses more power. Typical values range from **1 kΩ to 10 kΩ**, balancing speed and efficiency.

Values of the pull-up resistors are one of the factors that limit the maximum bus speed. I2C supports various **modes**, each with a defined maximum bus speed. Pull-up resistor values and other factors, such as bus capacitance, often limit the actual achievable speed. Devices are categorized by the modes they support, allowing theoretical operation at specific speeds. **High-Speed Mode** is backward compatible and temporarily enables faster communication using a special sequence. **Ultra-Fast Mode** is unidirectional (write-only) and modifies the standard I2C protocol to achieve even higher speeds.

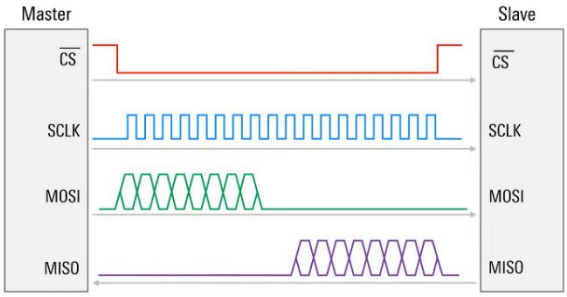
I2C Mode	Speed
Standard Mode	100 kbps
Fast Mode	400 kbps
Fast Mode Plus	1 Mbps
High Speed Mode	3.4 Mbps
Ultra Fast Mode	5 Mbps

2.3. SPI Protocol

SPI (Serial Peripheral Interface) is a 4-wire serial interface developed by Motorola in the 1980s. It offers faster data transfer than UART or I^2C and supports full-duplex communication. However, SPI lacks strict standards for frame formats, bit counts, or voltage levels. It's commonly used to transfer data between a smart controller and a peripheral device. Typical applications include sensors, displays, ADCs, DACs, real-time clocks, and wired game controllers.

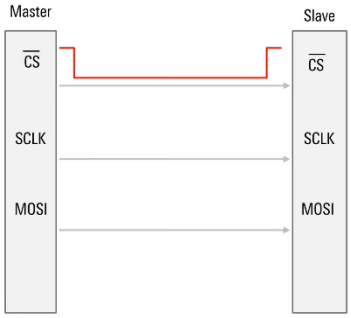
SPI is a **Master-Slave** protocol with one master (controller) and one or more slaves (peripherals). These are typically connected by four wires:

1. **Chip Select (CS):**
   The master pulls this line low to select the slave it wants to communicate with.
   At the end of communication, the master pulls CS back to high.
2. **Synchronous Clock (SCK):**
   Generated by the master to synchronize data transfer with the slave.
3. **MOSI (Master Out Slave In):**
   Used to send bits from the master to the slave.
4. **MISO (Master In Slave Out):**
   Used to send bits from the slave to the master, if the slave has data to send.



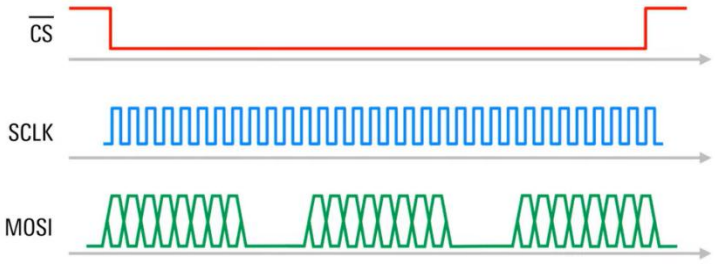
While these are the terms used here, other names may be used in SPI-based systems. In some special cases, not all four wires are required.

1. **Chip Select (CS):** also called **Slave Select**, is typically active low $(\overline{CS})$ can manage addressing with either a single line or multiple CS lines
   When the master pulls the CS line low, communication starts. Then master prompting the slave to listen for the clock (SCK) and data (MOSI) signals.
   Once communication ends, CS is returned to its idle state.
   CS simplifies slave addressing by not requiring explicit addresses, unlike I2C or CAN bus.

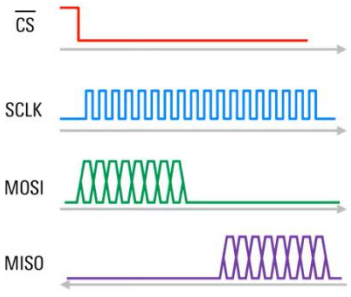


2. **SCLK(Serial Clock):** after the master pulls CS low to start communication, it generates the SCLK (Serial Clock) signal. This eliminates the need for slaves to have their own clocks. SPI doesn't have a standardized clock rate, but typical speeds are in the MHz range, making it faster than protocols like UART or I2C. The clock indicates when data should be sampled, with one bit read per clock cycle. The clock can idle at either a low or high state, and data can be sampled on either the rising or falling edge. Further details will be covered later.

3. **MOSI (Master Out Slave In):** once the CS line is pulled low and the clock starts, MOSI is used to send data from the master to one or more slaves. Data is typically sent as 8-bit bytes, either with the least or most significant bit first. The number of bytes sent is implementation-dependent, and SPI allows multiple bytes to be sent sequentially. The CS line can be held low while sending multiple bytes, but this behavior may vary depending on the implementation.

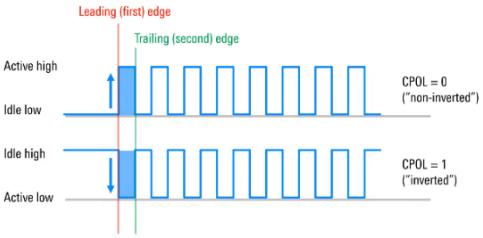


4. **MISO (Master In Slave Out):** is used to send data from the slave to the master. In some configurations, MISO is optional if the slave device is only a receiver, such as a display, which requires fewer than four wires. Since only the master generates the clock signal, it must know how many bits the slave will send to provide the correct number of clock cycles. This number (or other commands) will be determined by querying the slave.



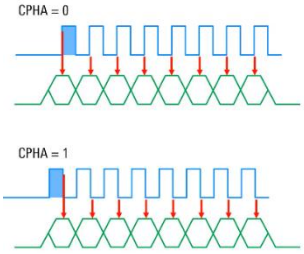
After covering the basics of SPI, it is essential to address **CPOL (Clock polarity)**, **CPHA (Clock Phase)**, and **Multi-Slave Configurations** for proper communication and synchronization.

- **CPOL (Clock polarity):** refers to the idle state of the clock signal generated by the master.
  - **CPOL = 0** : when the clock is low in idle it's **active high** and leading edge is a **rising edge↑**
  - **CPOL = 1** : when the clock is high in idle, it's **active low** and leading edge is a **falling edge↓**



The leading edge of a clock pulse is the first edge, and the trailing edge is the second.

- **CPHA (Clock Phase):** refers to when data is sampled during communication. In SPI, a receiver can sample data on either the leading (first) edge or the trailing (second) edge of the clock pulse.
  - **CPHA = 0** means data is sampled on the leading (first) edge
    (Not starting sampling, but data availability)
  - **CPHA = 1** means data is sampled on the trailing (second) edge.



(In both cases, the examples assume the clock has a polarity of zero / idle low / CPOL0.)

SPI Modes: The two types of clock polarity and two types of clock phase combine to create four possible SPI modes.
All connected SPI devices must operate in the same mode, which may be fixed in hardware or configured by the user.

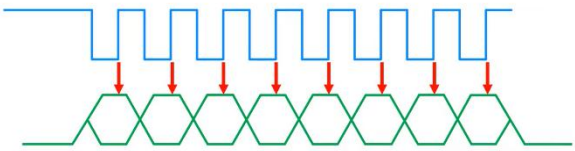
Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

The mode can often be identified by analyzing SPI waveforms.
For instance, if the clock signal idles high (CPOL = 1) and data is read on the trailing (second) edge (CPHA = 1), the communication uses SPI mode 3.

CPOL = 1 (clock idle high)

CPHA = 1 (read on trailing edge)

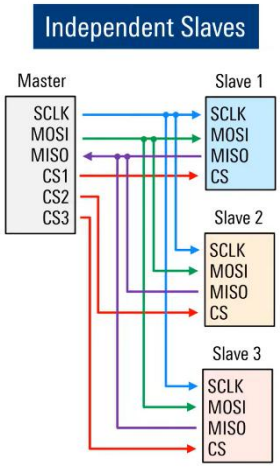
Therefore, mode = 3



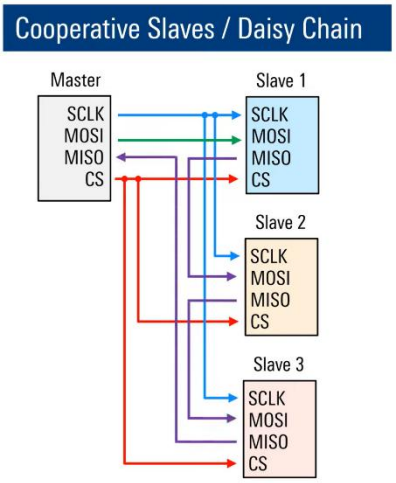
Among these, mode 0 is the most commonly used in SPI implementations.

- **Multi-Slave Configurations:** In SPI, multiple slaves can be managed using two configurations:

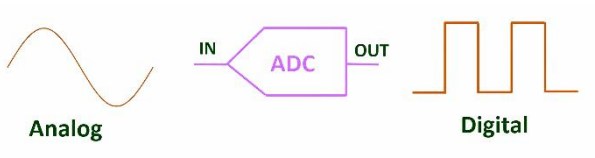
  - **Independent Slaves:**
    - Each slave has its own CS line.
    - The master pulls the specific CS line low to communicate with a slave, while other lines (SCLK, MOSI, and MISO) are shared.
    - This method is simple but becomes challenging to scale due to the need for separate CS lines for each slave.



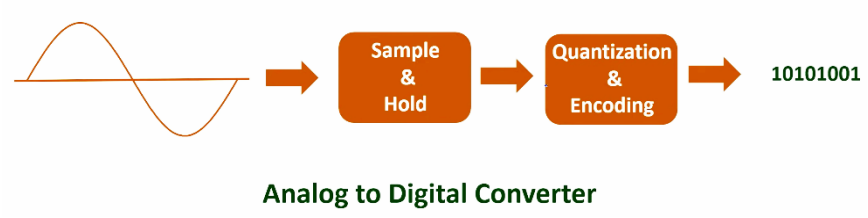
  - **Cooperative Slaves (Daisy Chain):**
    - A single CS line is shared among all slaves.
    - MOSI data is sent to the first slave, which passes it along the chain to subsequent slaves via clock cycles, connecting each slave's MISO to the next's MOSI.
    - Data from the last slave is routed back to the master.
    - This method minimizes CS lines but is more complex and may not be supported by all SPI devices.



3. Analog-to-Digital Converters (ADC)

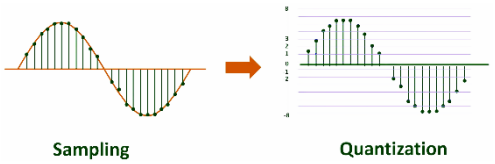


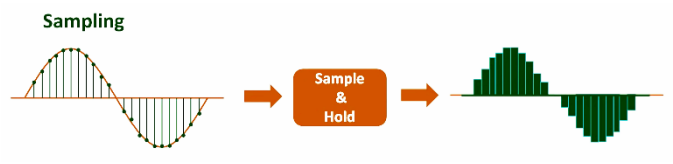
An Analog-to-Digital Converter (ADC) transforms continuous analog signals, such as temperature, sound, or pressure, into digital values that computers and microcontrollers can process. This conversion is crucial because digital signals are less affected by noise, easier to store, and more efficient to analyze. ADCs enable real-world data to be used in digital systems, supporting applications like sensor interfacing, audio processing, and automation, making them indispensable in modern electronics.
The ADC process consists of three main steps: **sampling**, **quantization**, and **encoding**.



1. **Sampling & Hold**

   - Sampling measures the analog signal's value at regular intervals, called the sampling rate.
   - According to the **Nyquist Theorem**, the sampling rate must be at least twice the maximum frequency of the signal to avoid inaccuracies like aliasing.
   - An analog signal varies continuously, so sampling **holds** its value steady enough for the ADC to process it accurately during quantization and encoding.





2. **Quantization**

   - means assigning the sampled analog value to the closest value in a fixed set of discrete levels that the ADC can represent and **resolution** of the ADC determines the number of these levels.
   - **Resolution** defines the smallest change in the input signal that the ADC can detect, determined by the number of (n)bits in the ADC. It is calculated as:

$$Resolution\ (Step\ Size) = \frac{Full\ Scale\ Range\ (FSR)}{2^n} = \frac{Vref}{2^n}$$

For example, a 3-bit ADC with a reference voltage of 10V has $2^3 = 8$ levels, giving a resolution of $\frac{10}{2^3} = 1.25v$. This means that the smallest detectable change in the input voltage of ADC is 1.25V. If the input voltage increases or decreases by less than 1.25V, the digital output remains unchanged, introducing **quantization error**.

The digital value output by the ADC can be calculated as:

$$ADC\ Value = \frac{Vin \cdot 2^n}{Vref}$$

3. **Encoding**
   The quantized levels are converted into a binary format for digital processing. The digital output forms a "staircase" transfer function, where each step corresponds to a specific quantized level. Higher resolution results in smaller steps, making the output closer to the actual signal.

3.1. External ADC-0831N

The ADC0831-N provides 8-bit resolution, converting analog signals into 256 discrete levels. It supports differential and single-ended inputs, improving noise immunity and flexibility in diverse applications. Communication is handled via a 3-wire serial interface, with data output in **MSB-first** format.

Its **built-in sample-and-hold circuit** ensures stable signal conversion, even when working with rapidly changing analog inputs. The compact design and low power consumption make it ideal for battery-operated devices.

The ADC0831-N uses **successive approximation** to convert analog signals into digital form. The reference voltage defines the input range and resolution:

$$Resolution\ (Step\ Size) = \frac{Vref}{2^n}$$

For example, with Vref=5V, the resolution is approximately 19.53 mV per bit.

The digital output is transmitted in **MSB-first** order, synchronized with the clock signal provided by the master device. Differential inputs further enhance the ADC's ability to reject noise and improve accuracy.

Pin Configuration

1. **CS:** Chip Select, activates the ADC for communication.
2. **IN+:** Positive analog input.
3. **IN-:** Negative analog input, used in differential mode.
4. **GND:** Ground reference.
5. **VCC:** Power supply, typically 5V.
6. **CLK:** Clock input for synchronizing data transfer.
7. **DOUT:** (MISO) Digital data output (MSB-first).
8. **VREF:** Reference voltage input defines the ADC's full-scale range.

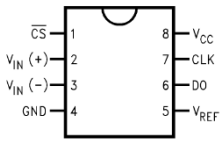


Figure 4. ADC0831-N Single Differential Input PDIP Package (P) Top View

Application, Use and Characteristics:

ADC0831-N is often used to interface with sensors and peripherals in systems requiring accurate analog-to-digital conversion. By pulling the **CS pin low**, the master device activates the ADC, and conversion begins. Data is transmitted over the **DOUT** line, synchronized with the clock signal on the **CLK** pin with maximum clock speed($\mathbf{f_{CLK}}$) up to 400 kHz.

AC Characteristics

The following specifications apply for V_{CC} = 5V, $t_r = t_f$ = 20 ns and 25°C unless otherwise specified.

Parameter		Conditions	Typ(1)	Tested Limit(2)	Design Limit(3)	Limit Units
f_{CLK}, Clock Frequency	Min			10		kHz
	Max				400	kHz
t_C, Conversion Time		Not including MUX Addressing Time		8		$1/f_{CLK}$
Clock Duty Cycle(4)	Min				40	%
	Max				60	%
t_{SET-UP}, $\overline{CS}$ Falling Edge or Data Input Valid to CLK Rising Edge					250	ns
t_{HOLD}, Data Input Valid after CLK Rising Edge					90	ns
t_{pd1}, t_{pd0}—CLK Falling Edge to Output Data Valid(5)		C_L=100 pF				
		Data MSB First	650		1500	ns
		Data LSB First	250		600	ns
t_{1H}, t_{0H}—Rising Edge of CS to Data Output and SARS Hi–Z		C_L=10 pF, R_L=10k (See TRI-STATE Test Circuits and Waveforms)	125		250	ns
		C_L=100 pf, R_L=2k		500		ns
C_{IN}, Capacitance of Logic Input			5			pF
C_{OUT}, Capacitance of Logic Outputs			5			pF

(1) Typicals are at 25°C and represent most likely parametric norm.
(2) Tested limits are ensured to TI's AOQL (Average Outgoing Quality Level).
(3) Ensured but not 100% production tested. These limits are not used to calculate outgoing quality levels.
(4) A 40% to 60% clock duty cycle range insures proper operation at all clock frequencies. In the case that an available clock has a duty cycle outside of these limits, the minimum, time the clock is high or the minimum time the clock is low must be at least 1 µs. The maximum time the clock can be high is 60 µs. The clock can be stopped when low so long as the analog input voltage remains stable.
(5) Since data, MSB first, is the output of the comparator used in the successive approximation loop, an additional delay is built in (see ADC0838-N Functional Block Diagram) to allow for comparator response time.

Timing: Important point is to ensure reliable data transfer when fetching data from the **ADC0831-N**, the master device needs to account for **both Tpd(Propagation Delay)** and **Tsetup (Setup Time)**.

- **Tpd (Propagation Delay):**
  The time delay between a clock edge and the appearance of valid data on the output.
- (between 650ns – 1500ns)

- **Tsetup (Setup Time):**
  The minimum time the data must remain stable before the next clock edge for accurate sampling. (250ns)

Timing Diagrams

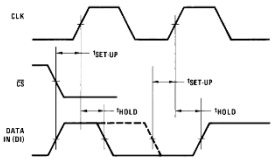


Figure 15. Data Input Timing

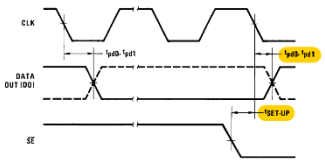


Figure 16. Data Output Timing

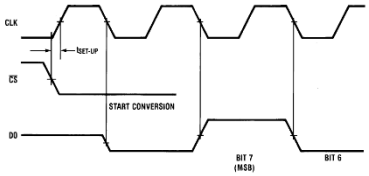
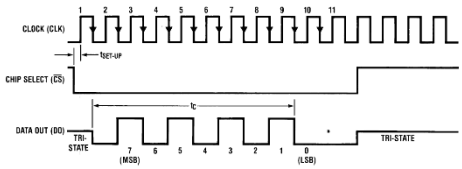


Figure 17. ADC0831-N Start Conversion Timing



*LSB first output not available on ADC0831-N.

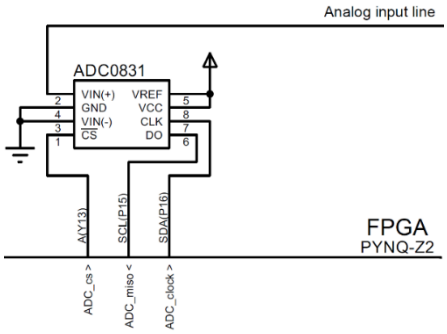
Figure 18. ADC0831-N Timing

3.2. Implementation of ADC0831-N on FPGA

In this project, I used the **ADC0831-N** as an external ADC to measure the analog output value of the power supply section which is connected to an FPGA configured as the master device in the SPI communication protocol.

Configuration Details:

- **Clock Frequency (Fclk):** 250 kHz
- **Propagation Delay (Tpd):** 800 ns
- **Setup Time (Tsetup):** 250 ns
- **SPI Mode: 0** (CPOL = 0, CPHA = 0)zz



FPGA Master Unit for ADC0831:

As discussed, the ADC0831 works as an SPI slave, which requires an FPGA-based master to communicate with it. The master unit generates SPI signals, reads digital data from the ADC, and stores the output values. Key aspects need to be considered:

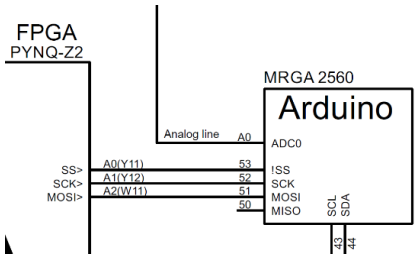
- The FPGA operates at **125 MHz** (5ns), and for the ADC0831, the clock is reduced to **250 kHz** (4us), meaning **500 FPGA clock cycles** equal one ADC clock cycle. (125Mhz/250Khz)
- **Tsetup**: After pulling **CS** low, a delay of **250 ns** (31×8ns(FPGA clk) ≈ 248ns) is required before starting the ADC clock.
- The **first clock** pulse is idle, serving as a synchronization step with no valid data for sampling.
- **Tpd**: After the first valid clock, wait for **800 ns** (100×8ns=800ns) before fetching valid data from the ADC.
- **Stop**: Once 8 bits are read, wait for one more clock cycle, then set **CS** and the clock back to the idle state.

Well commented code below described these matters:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ADC0831 is
    port(
        ref_clk:  in   std_logic;  --125MHZ (8ns) | (FPGA clock)
        spi_clk:  out  std_logic;  --250KHZ (4us)  | to ADC
                                   --means each 500 ref_clk is one spi_clk
        spi_cs:   out  std_logic;  --chip select active low
        spi_miso: in   std_logic;  --Pin Do in ADC
        spi_data: out  std_logic_vector(7 downto 0);
                                   --8 bit space for saving sampled data readed from ADC to it
        ADCRead:  in   std_logic;  --trigger / Sampling key (1 means sapmling|0 means idel)
        busy:     out  std_logic   --status (1 busy) for sampling or not
    );
end ADC0831;

architecture Behavioral of ADC0831 is
    type state is (IDLE, Tsetup, Read1, Read2, Stop);        --making suitable pulse to read from ADC
    signal FSM: state := IDLE;                               --data type: state (FSM changable)
    signal spi_clk_counter: integer range 0 to 500 := 0;     --data type: integer (250 used)
    signal temporaryData: std_logic_vector(8 downto 0) := (others => '0');  --9 gets but 8bit is valid data(first is invalid)
    signal dataIndex: integer range 0 to 8 := 8; --9th index is for timing setting so it's not valid
    signal spi_clk_int: std_logic := '0';

begin
    spi_clk <= spi_clk_int;                                  --spi_clk=0
    process(ref_clk)                                         --process sensitive to ref_clk (125MHz)
    begin
        if rising_edge(ref_clk) then                         --working on rising(positive) edge
                                                             --/ synchrone data with ref_clk
            case FSM is
                when IDLE =>                                 --not working
                    spi_cs <= '1';
                    spi_clk_int <= '0';
                    spi_clk_counter <= 0;
                    dataIndex <= 8;
                    busy <= '0';                             --make 'one bit' zero
                    temporaryData<= (others => '0');         --make 0 all vector's indexes
                    if ADCRead = '1' then                    --when sampling key is pressed initialize
                        FSM <= Tsetup;
                        spi_cs <= '0';
                        busy <= '1';
                    end if;

                when Tsetup =>                               --next time process running comes to Tsetup
                    spi_clk_counter <= spi_clk_counter + 1;  --spi_clk_counter ++
                    if spi_clk_counter = 31 then             --(1/125Mhz) *31 = 248ns wait for Tsetup
                        FSM <= Read1;                        --next time FSM goes to Read1
                        spi_clk_int <= '1';                  --spi_clk = 1
                        spi_clk_counter <= 0;                --initialize
                    end if;

                when Read1 =>                                --in first positive clock no data is valid!
                    spi_clk_int <= '1';                      --spi_clk = 1
                    spi_clk_counter <= spi_clk_counter + 1;  --spi_clk_counter ++
                    if spi_clk_counter = 250 then            --(250*8ns=2us) making first positive spi_clk=1
                        spi_clk_int <= '0';                  --spi_clk = 0
                        spi_clk_counter <= 0;
                        FSM <= Read2;                        --next time FSM goes to Read2
                    end if;

                when Read2 =>
                    spi_clk_counter <= spi_clk_counter + 1;
                    if spi_clk_counter = 250 then            --(250 * 8ns = 2us) time for half spi_clk pulse
                        spi_clk_int <= not spi_clk_int;      -- making spi_clk +-
                        spi_clk_counter <= 0;
                    end if;
                    if spi_clk_int = '0' and spi_clk_counter =100 then--Tpd(after 8ns*100=800ns data is Valid)
                        temporaryData(dataIndex) <= spi_miso;--MISO(data) goes in 8th index of temporaryData
                        if dataIndex = 0 then
                            FSM <= Stop;
                            spi_clk_counter <= 0;
                        else
                            dataIndex <= dataIndex - 1;
                        end if;
                    end if;
                when Stop =>                                 --one clk more for CS and data for stop
                    spi_clk_counter <= spi_clk_counter + 1;  --spi_clk_counter ++
                    if spi_clk_counter = 250 then            --waint for one spi_clk more
                        spi_clk_int <= not spi_clk_int;
                        spi_clk_counter <= 0;
                        if spi_clk_int = '0' then
                            spi_cs <= '1';                --9data gets but 8bit is valid data(first is invalid)
                            spi_data <= temporaryData(7 downto 0);  --9th data is for time setting/invalid!
                            FSM <= IDLE;                          --next time FSM goes to IDEL
                        end if;
                    end if;
            end case;
        end if;
    end process;
end Behavioral;
```

4. FPGA SPI Master Unit for Arduino

As previously discussed, the FPGA Master Unit facilitates communication with the external **ADC0831**, where the received data is stored in `spi_data <= temporaryData(7 downto 0)` and then transferred to the **External SRAM** (will be detailed later in the **Top Module** section).



Another unit created by FPGA is **"Master SPI unit"**, responsible for communication between the **FPGA** and **Arduino**. Main duty of this unit is sending stored data from SRAM to Arduino through SPI protocol handled in FPGA. This unit still requires some considerations based on the Arduino operating as the SPI slave, as outlined below:

- **Data Order: MSB First**
- **Clock Frequency: 4 MHz**
- **SPI Mode: Mode 0**
- **Data Frame Size: 8-bit frames**

While the well-commented code below describes the specifications of the **Master SPI unit** on the **FPGA**, the connection of this unit with the top module and other sections, as well as the configuration of the Arduino at the register level, will be discussed in more detail in another section:

```
// Function to initialize SPI in slave mode
void initSPI() {
  DDRB &= ~(1 << DDB4);      // Set PB4 (SPI MISO) output
  SPCR = (1 << 7)            // Enable SPI SPCR (SPIE=1)
       | (1 << 6);           // Enable SPI (SPE=1)
  SPCR &= ~((1<<5)           //MSB first (DORD=0)
       | (1<<MSTR)           //SLAVE (MSTR=0)
       | (1<<3)              //CPOL=0
       | (1<<2)              //CPHA=0
       | (1<<SPR1)           //SPI CLK (SPR1,SPR0=0)
       | (1<<SPR0));         //SPI CLK=16Mhz/4=4Mhz
}
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SPI_Master is
    port(
        sys_clk     : in    std_logic;      --125Mhz
        spi_mosi    : out   std_logic;      --MOSI
        spi_sck     : out   std_logic;      --SPI CLOCK
        spi_ss      : out   std_logic;      --CS
        data_in     : in    std_logic_vector(7 downto 0); --DATA
        start_trans : in    std_logic       --sampling key
    );
end SPI_Master;

architecture Behavioral of SPI_Master is
    type state_type is (IDLE, LOAD, SEND, COMPLETE); --FSM / cases
    signal current_state : state_type := IDLE;
    signal bit_index     : integer range 0 to 8 := 8;--first Clock has no data
    constant CLOCK_DIV   : integer := 31;            --125Mhz/4Mhz(F-SPI)=31
    signal clk_counter   : integer range 0 to 30:= 0;
    signal sck_pulse     : std_logic := '1';
    signal extended_data: std_logic_vector(8 downto 0);
begin
    extended_data(7 downto 0) <= data_in;
    extended_data(8) <= '0';
    process(sys_clk)
    begin
        if rising_edge(sys_clk) then
            case current_state is
                when IDLE =>                    --first condition / do nothing
                    spi_ss    <= '1';
                    spi_sck   <= '0';
                    spi_mosi  <= '0';
                    sck_pulse <= '0';
                    if start_trans = '1' then --sampling key pressed
```

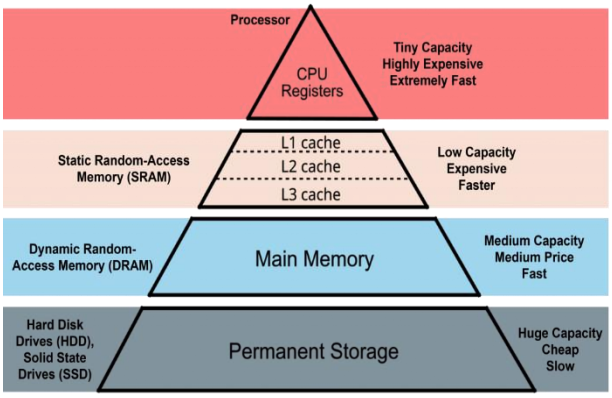
```
                    current_state <= LOAD;
                end if;

            when LOAD =>
                spi_ss    <= '0';
                bit_index <= 8;
                spi_mosi  <= extended_data(bit_index); --MOSI=extended_data(8)
                clk_counter <= 0;
                sck_pulse <= '0';
                current_state <= SEND;

            when SEND =>
                clk_counter <= clk_counter + 1;
                spi_mosi <= extended_data(bit_index);   --8 (9th data)= MSB
                  if clk_counter = (CLOCK_DIV / 2) then --Making Half clock pulse
                    sck_pulse <= not sck_pulse;         --after ~15 CLOCK
                    spi_sck <= sck_pulse;
                    clk_counter <= 0;
                    if sck_pulse = '0' then
                    --at "Process" changes will be implimented on next cycle
                        bit_index <= bit_index-1;
                        if bit_index = 0 then
                            current_state <= COMPLETE;
                        end if;
                    end if;
                  end if;
            when COMPLETE =>
                spi_ss <= '1';
                current_state <= IDLE;
        end case;
      end if;
  end process;
end Behavioral;
```

5. Introduction to Memories

Memories are fundamental to digital systems, serving as storage units for data and instructions. Organized into addressable words of fixed widths, memories vary in **size**, **speed**, **volatility**, and **functionality**, offering solutions tailored to specific applications. This report provides an overview of memory hierarchy, their classifications, and characteristics, emphasizing Static Random Access Memory (SRAM) and its implementation in a project context.

Memory Hierarchy: refers to the structured arrangement of memory types in a system, organized by speed, cost, and capacity. It is represented as a pyramid:

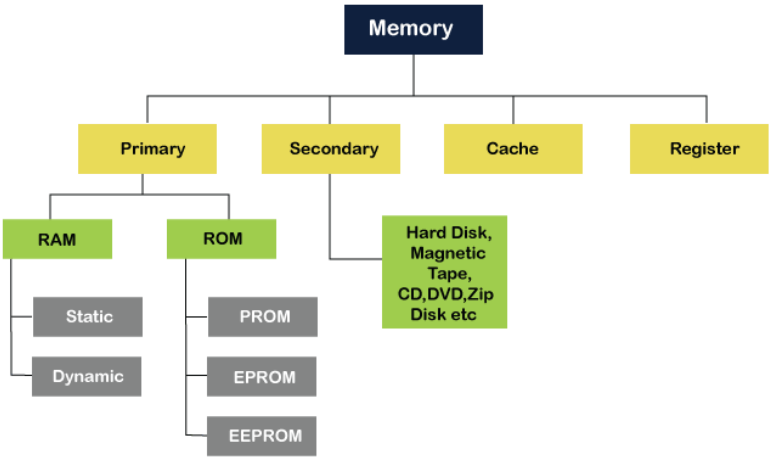
1. **Registers:** Fastest, most expensive, smallest capacity; located within the CPU for immediate data access.
2. **SRAM / Cache (L1, L2, L3):** Slightly slower than registers but faster than other memory types; holds frequently accessed data close to the CPU.
3. **Main Memory (DRAM):** Larger capacity, slower speed; off-chip memory used for regular data storage.
4. **Secondary Storage (SSDs, HDDs):** Much larger capacity, significantly slower; used for long-term data storage.
5. **Remote Storage (Cloud):** Largest capacity, slowest speed, accessed over the network.



As you move down the hierarchy, memory becomes cheaper per bit, larger in capacity, but slower in access speed. This structure optimizes performance and cost efficiency in computing systems.

Memory classifications: refers to the systematic categorization of computer memory into distinct types based on characteristics such as **functionality** (primary, secondary), **volatility** (volatile, non-volatile), **access type** (random, sequential), **technology** (SRAM, DRAM), **speed** (cache, main memory, storage), **location** (on-chip, off-chip), and **portability** (fixed, removable).

These classifications help in understanding how different memory types of function, interact, and serve various roles in a computing system.



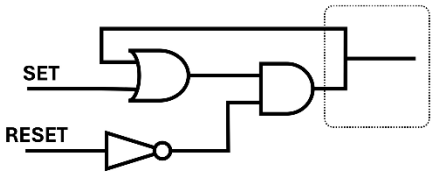
This report emphasizes Static Random Access Memory (**SRAM**), detailing its role and implementation in a project context, along with key aspects like speed, structure, and interfacing.

5.1. SRAM

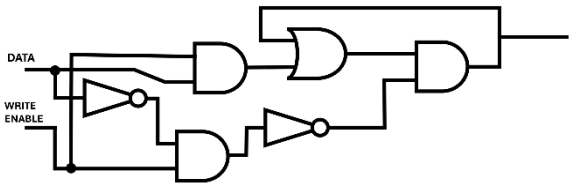
Static Random-Access Memory (SRAM) is a type of volatile memory that uses bistable latching circuits to store data. Unlike DRAM, it does not require periodic refreshing, making it faster and more reliable for temporary data storage. SRAM is commonly used in cache memory and embedded systems due to its low latency and high-speed performance, though it is more expensive and consumes more space compared to DRAM.

Memory refers to the ability to retain information, achieved using **AND-OR Latches** where one input sets the value to one, and another resets it to zero.

By combining circuits, we can create a latch capable of "remembering" either 0 or 1. This circuit stores one bit of information, which can be determined by reading the value from its output wire.



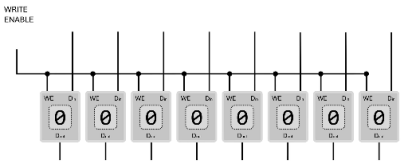
If both inputs are set to one simultaneously, the circuit behaves unpredictably. To solve this, with the addition of a few extra logic gates, an additional input called **WR Enable** is introduced, allowing control over when the value should be stored. This input, ensures a more reliable and intuitive mechanism for storing data.

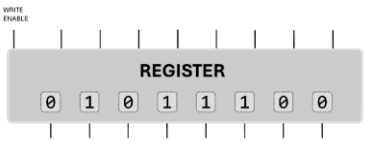


When the Write Enable input is set to zero the value in the Data signal is completely ignored and nothing will happen, however when the Write Enable input is set to one the circuit will retain whatever value is being passed to the Data input

This circuit can abstract into a component known as a **Gated-Latch**. There are several other ways to create latches. A gated latch is capable of storing one bit of information if its right enable input is set to zero no action will occur but if the WR enable input is set to one it will retain whatever bit of information is passed through the data input.

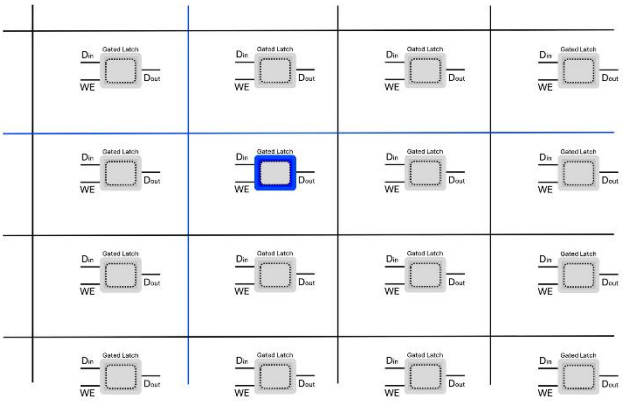
To store more than a single bit, multiple latches can be combined. For example, eight latches can store one byte of information, with their **Write Enable** inputs unified to allow simultaneous updates. This arrangement is called a **Register**. (registers come in different sizes)





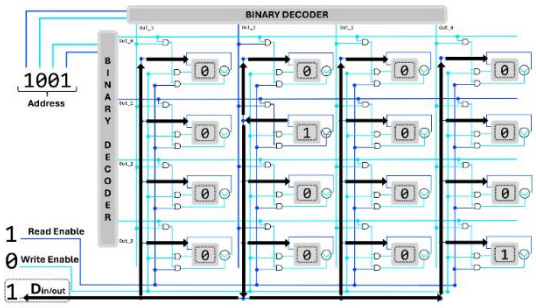
1 byte = 8 bits

To store larger amounts of data, latches are arranged in a 4x4 matrix, creating 16 latches in total. Each latch can be uniquely identified by using four wires for rows and four for columns. Activating one wire from both the row and column simultaneously selects a specific latch in the matrix.



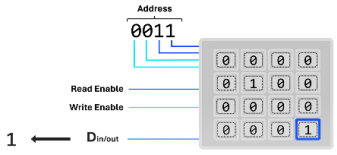
With decoder for rows and columns we can input an address to select a specific latch in the Matrix. Each latch possesses its unique address which gives rise to the term **memory address.**

By adding logic gates, read and write enable signals, and a shared data line, we create a memory matrix where each latch can independently store, update, and output values based on its address, minimizing the number of wires while ensuring precise read and write operations.

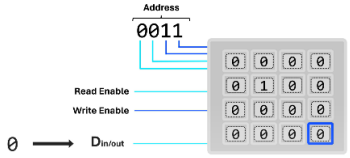


With the address input we can select any of the internal latches, now:

- By using the **Read** enable signal we make the data line output the current value of the selected latch.

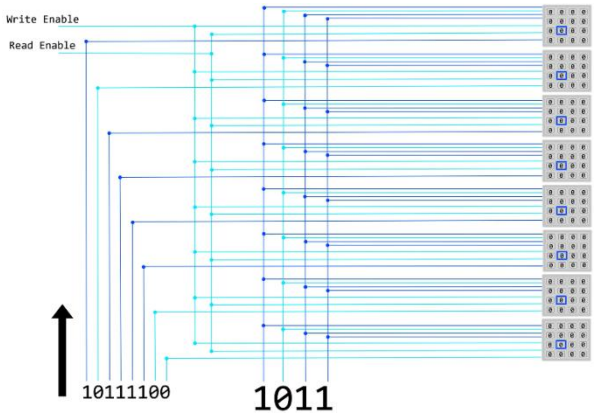


- Conversely if we use the **WR Enable** signal we can override the value in the selected latch using the data line as input



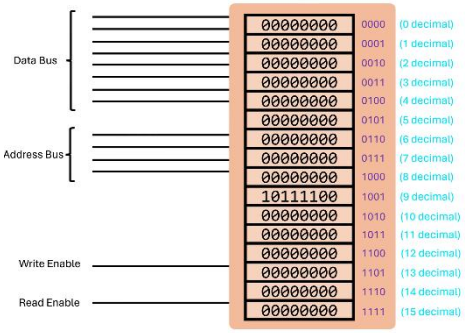
To store bytes instead of bits, eight matrices are used, each representing one bit of a byte. These matrices share the same address input, allowing the selection of corresponding latches across all matrices using the same address. The **read enable** and **write enable** inputs are also shared to read or write an entire byte simultaneously. Each matrix has its own data line for accessing individual bits.

For example, to write a byte to memory, the bits making up the byte are input through the data lines, and the **write enable** signal is activated. This stores each bit in its corresponding latch at the specified address. Later, to read the byte, the address is input, and the **read enable** signal is activated, causing the data lines to output the byte stored at that address.



This abstraction is known as **Random Access Memory (RAM)** because any byte in the list can be accessed directly by providing its address.

To manage memory, sets of wires are defined as a **bus**. The width of the **address bus** determines how much memory can be accessed. Generally, an **n-bit address bus** can address 2^n bytes. For instance, a 32-bit address bus can address up to 4 billion bytes, which is why 32-bit computers are limited to 4 GB of RAM.



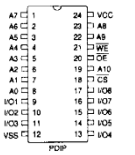
The discussed memory is **Static RAM (SRAM)**, which retains data as long as it remains powered, eliminating the need for periodic refresh cycles required by **Dynamic RAM (DRAM)**. While SRAM offers superior speed, its reliance on numerous logic gates and transistors makes it significantly more expensive to manufacture.

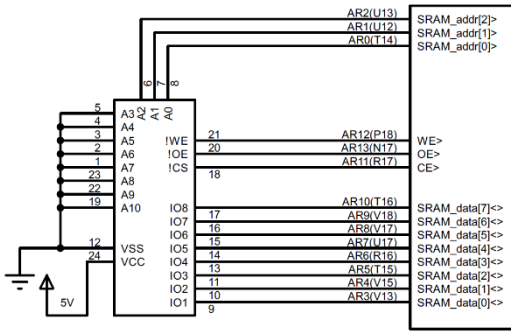
5.2. implementation of SRAM HY6116ALP

The HY6116ALP is a 2K × 8-bit static RAM (SRAM), meaning it can store **2048 bytes** (8-bit each), with an **11-bit address bus** (A0–A10) and an **8-bit data bus** (I/O0–I/O7). This makes it suitable for high-speed, temporary data storage in digital systems. The **HY6116ALP** uses **parallel communication**, meaning data and addresses are transferred via multiple lines **simultaneously**. It's faster than serial but requires more pins and wiring, and unlike DRAM, data remains stored as long as power is supplied. While for control are active low as blow:

- **CS**: Chip Select must be **LOW (0)** to enable the SRAM.
- **WE**: Write Enable must be **LOW (0)** to write data.
- **OE**: Output Enable must be **LOW (0)** to read data

PIN CONNECTION





Since the project involves storing **8-bit ADC data**, only 3 address lines are used, allowing up to 8 different storage locations. The remaining address lines must be connected to ground (0).

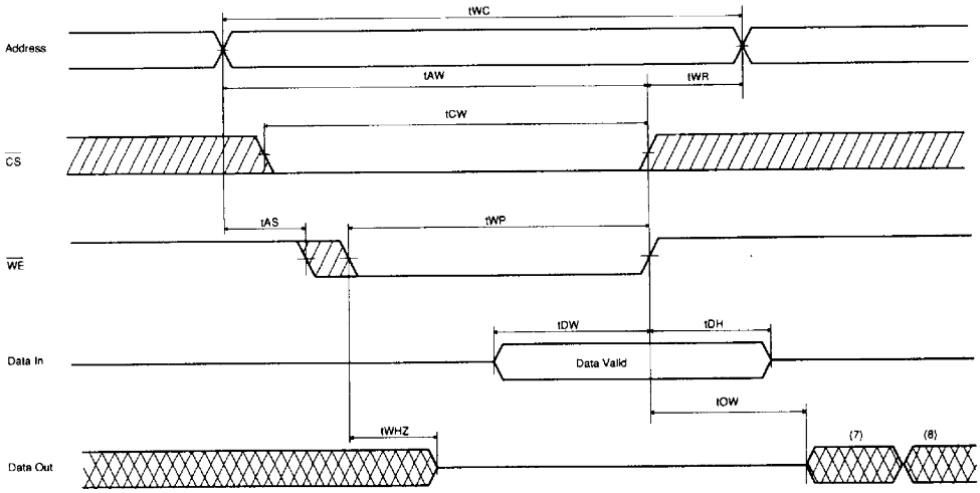
Writing Data to SRAM:

- Set **CS = 0** and **WE = 0**.
- Send the address on the **address bus** (A0–A2).
- Place **ADC data (8-bit)** on the **data bus**.
- Maintain **tDW ≈ 40ns** for data validity.
- Set **WE = 1** after writing.

WRITE CYCLE

10	tWC	Write Cycle Time	85	-	100	-	120	-	150	-	ns
11	tCW	Chip Select to End of Write	60	-	65	-	70	-	90	-	ns
12	tAW	Address Valid to End of Write	70	-	80	-	105	-	120	-	ns
13	tAS	Address Set-up Time	0	-	0	-	0	-	0	-	ns
14	tWP	Write Pulse Width	55	-	60	-	70	-	80	-	ns
15	tWR	Write Recovery Time	0	-	0	-	0	-	0	-	ns
16	tWHZ	Write to High-Z Output	0	30	0	30	0	35	0	40	ns
17	tDW	Data to Write Time Overlap	30	-	30	-	35	-	40	-	ns
18	tDH	Data Hold from Write Time	0	-	0	-	0	-	0	-	ns
19	tOW	Output Active from End of Write	10	-	10	-	10	-	10	-	ns

WRITE CYCLE 2 ($\overline{OE}$ Low Fixed)



Reading Data from SRAM:

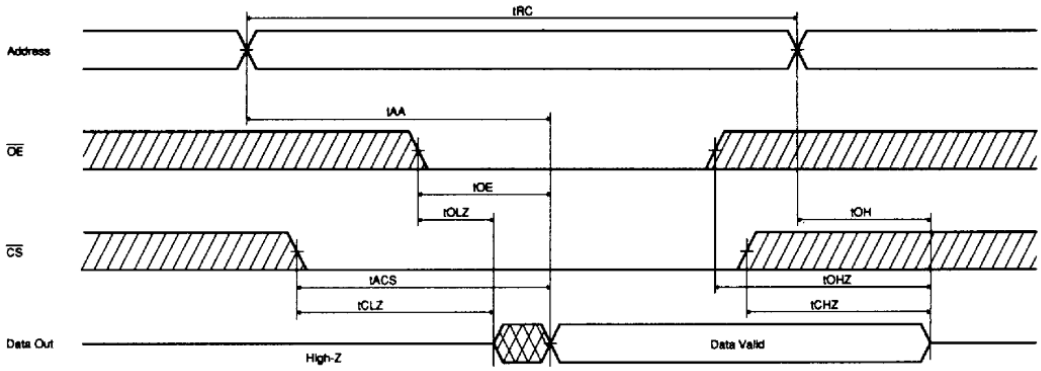
- Set **CS = 0** and **OE = 0**.
- Send the address on the **address bus**.
- Wait **tACS ≈ 90ns** for valid data.
- Read the data from the **data bus**.
- Set **OE = 1** after reading.

AC CHARACTERISTICS

(T_A= 0°C to 70°C, V_{CC}= 5V ± 10%, unless otherwise noted.)

#	SYMBOL	PARAMETER	-85		-10		-12		-15		UNIT
			MIN.	MAX.	MIN.	MAX.	MIN.	MAX.	MIN.	MAX.	
	READ CYCLE										
1	tRC	Read Cycle Time	85	-	100	-	120	-	150	-	ns
2	tAA	Address Access Time	-	85	-	100	-	120	-	150	ns
3	tACS	Chip Select Access Time	-	85	-	100	-	120	-	150	ns
4	tOE	Output Enable to Output Valid	-	45	-	50	-	55	-	60	ns
5	tCLZ	Chip Select to Low-Z Output	10	-	10	-	10	-	10	-	ns
6	tOLZ	Output Enable to Low-Z Output	10	-	10	-	10	-	10	-	ns
7	tCHZ	Chip Disable to High-Z Output	0	40	0	40	0	40	0	50	ns
8	tOHZ	Output Disable to High-Z Output	0	40	0	40	0	40	0	50	ns
9	tOH	Output Hold from Address Change	10	-	10	-	10	-	15	-	ns

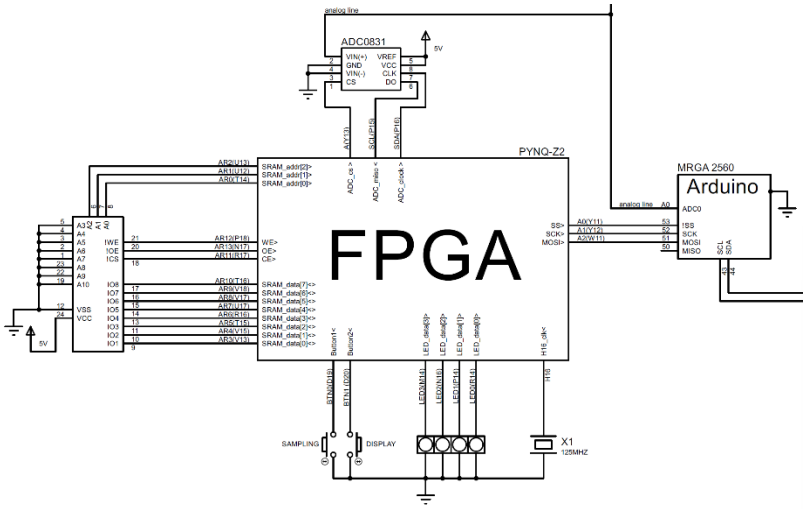
READ CYCLE 1



The way of communication with this SRAM plus its intersections with other parts of the project are discussed on the FPGA TOP MADULE provided blow.

6. FPGA TOP MADULE

The **top module** in this project serves as the main control unit that integrates the **ADC, SPI communication, and SRAM storage** within the FPGA. It manages data acquisition by triggering the ADC when **Button1(sampling)** is pressed, storing the sampled digital values into **SRAM**, and later retrieving and transmitting the stored data to an **Arduino via SPI** when **Button2(display)** is pressed. Additionally, it ensures proper synchronization between components, handles control signals for reading and writing operations, and provides visual feedback through **LED indicators** to display the number of stored samples and displays.



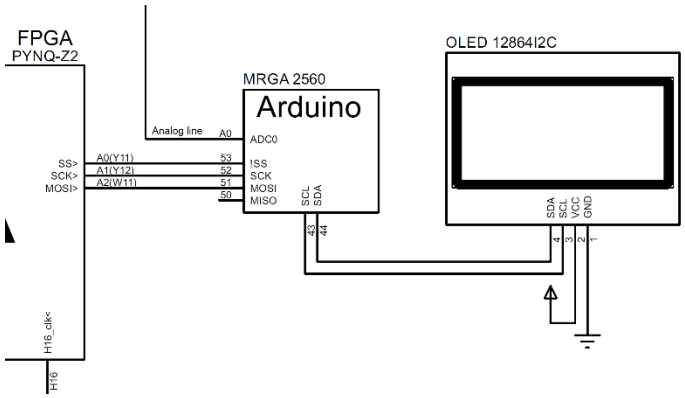
Ultimately FPGA, it is necessary to configure the I/O ports according to the board user manual. Below is your **top module code** with detailed comments, explaining how each section works:

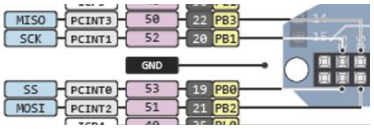
```
-- Import necessary libraries
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Entity definition for the top module (my_project)
entity my_project is
    port(
        H16_clk    : in  std_logic;  -- FPGA Clock (125 MHz)
        Button1    : in  std_logic;  -- Sampling trigger (ADC Start)
        Button2    : in  std_logic;  -- Display trigger (Send to Arduino)
        LED_data   : out std_logic_vector(3 downto 0);  -- LED indicators (sampling count)
        SRAM_data : inout std_logic_vector(7 downto 0); -- SRAM Data Bus (Bidirectional)
        SRAM_addr : out std_logic_vector(2 downto 0);  -- SRAM Address (3-bit)
        CE, WE, OE : out std_logic;  -- SRAM Control Signals
        MOSI, SCK, SS : out std_logic;  -- SPI interface (to Arduino)
        ADC_clock, ADC_cs : out std_logic;  -- SPI interface (to ADC)
        ADC_miso  : in  std_logic  -- ADC Data Line (MISO)
    );
end my_project;
-- Architecture defining internal behavior
architecture Behavioral of my_project is
    -- Instantiate SPI Master Component
    component SPI_Master is
        port(
            sys_clk     : in    std_logic;      -- System clock (125 MHz)
            spi_mosi    : out   std_logic;      -- SPI MOSI (Master Out)
            spi_sck     : out   std_logic;      -- SPI Clock
            spi_ss      : out   std_logic;      -- SPI Chip Select
            data_in     : in    std_logic_vector(7 downto 0); -- Data to transmit
            start_trans : in    std_logic       -- Start SPI transmission
        );
    end component;
    -- Instantiate ADC Component
    component ADC0831 is
        port(
            ref_clk:    in      std_logic;   -- FPGA Clock
            spi_clk:    out     std_logic;   -- SPI Clock for ADC
            spi_cs:     out     std_logic;   -- ADC Chip Select
            spi_miso:   in      std_logic;   -- ADC Data Line
            spi_data:   out     std_logic_vector(7 downto 0); -- ADC Output Data
            ADCRead:    in      std_logic;   -- ADC Sampling trigger
            busy:       out     std_logic    -- ADC Busy flag
        );
    end component;
    -- Internal Signals
    signal counter_1s: integer := 0;  -- Counter for debouncing
    signal write, read: std_logic_vector(1 downto 0) := "11";  -- FSM States
```

```
    signal index_w, index_r: integer := 0;  -- SRAM Write/Read Index
    signal write_cnt, read_cnt: integer := 0;  -- Delay Counters
    signal Data, spi_data, tmp_Data: std_logic_vector(7 downto 0);  -- Data Buffers
    signal Start, ADCRead, busy: std_logic := '0';  -- Control Signals
begin
    -- **Instantiate SPI Master and ADC Modules**
    SPI_master_unit: SPI_Master
        port map(H16_clk, MOSI, SCK, SS, Data, Start);
    ADC_unit: ADC0831
        port map(H16_clk, ADC_clock, ADC_cs, ADC_miso, spi_data, ADCRead, busy);

    -- **Main Process for Handling Sampling & Data Storage**
    process(H16_clk)
    begin
        if rising_edge(H16_clk) then
            Start <= '0';   -- Reset SPI start signal
            ADCRead <= '0'; -- Reset ADC trigger

            -- **Button Debounce Logic (500ms delay)**
            if counter_1s = 75000000 then
                -- **Handle Button1 (ADC Sampling)**
                if Button1 = '1' and busy = '0' and write = "00" then
                    SRAM_data   <= (others => 'Z');  -- Tri-state (prevent overwrite)
                    ADCRead     <= '1';  -- Trigger ADC Sampling
                    write       <= "01"; -- Move to Write State
                end if;

                -- **Handle Button2 (Data Display)**
                if Button2 = '1' and read = "00" then
                    read <= "01";  -- Set Read Flag
                    SRAM_data <= (others => 'Z');  -- Set SRAM to High Impedance
                    Data      <= (others => '0');  -- Reset Data Buffer
                end if;

                counter_1s <= 0;  -- Reset debounce counter
            else
                counter_1s <= counter_1s + 1;  -- Increment debounce counter
            end if;

            -- **Handle ADC Data Write to SRAM**
            if write = "01" and busy = '0' then
                tmp_Data <= spi_data;  -- Store ADC Data in Buffer
                write <= "10";  -- Proceed to Write State
            end if;
            if write = "10" then
                write <= "11";  -- Write Complete
                CE <= '0';  -- Enable SRAM
                WE <= '0';  -- Enable Write Mode
                OE <= '1';  -- Disable Read Mode
                SRAM_addr <= std_logic_vector(to_unsigned(index_w, 3)); -- Set Address (convert int to 3bit std)
                LED_data  <= std_logic_vector(to_unsigned(index_w, 3)); -- Update LEDs
                SRAM_data <= tmp_Data;  -- Store ADC Data to SRAM
                Start <= '1';  -- Notify Arduino (Trigger Internal ADC)
                Data  <= x"73";  -- Send ASCII 'S' to Indicate Sampling
                index_w <= index_w + 1;  -- Increment Write Index
                if index_w = 7 then index_w <= 0;  -- Reset at 8 samples
            end if;
            elsif  write = "11" then
                write_cnt <= write_cnt + 1;
                if write_cnt = 6 then      -- Wait for Data Validation (tDW:6*8ns=48ns)
                    write_cnt <= 0;
                    write <= "00";
                    CE <= '1';
                    WE <= '1';
                    OE <= '1';
                end if;
            end if;

            -- **Handle SRAM Read (Sending Data to Arduino)**
            if read = "01" then
                read <= "10";
                CE <= '0';  -- Enable SRAM
                WE <= '1';  -- Disable Write Mode
                OE <= '0';  -- Enable Read Mode
                SRAM_addr <= std_logic_vector(to_unsigned(index_r, 3)); -- Read Address
                LED_data  <= std_logic_vector(to_unsigned(index_r, 4)); -- Update LEDs
                index_r   <= index_r + 1;  -- Increment Read Index
                if index_r = 7 then index_r <= 0;  -- Reset at 8 samples
            end if;

            elsif read = "10" then
                read_cnt <= read_cnt + 1;
                if read_cnt = 12 then  -- Wait for Data Validation (tACS:12*8ns=96ns)
                    read_cnt <= 0;
                    read <= "00";
                    Start <= '1';  -- Send Data to Arduino
                    Data  <= SRAM_data;  -- Transfer Data
                    CE <= '1'; WE <= '1'; OE <= '1';  -- Disable SRAM
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

7. Arduino Implementation

The **Arduino Mega 2560** serves as the project's processing unit, managing **SPI communication with the FPGA, internal ADC measurements, and OLED display output**. It responds to FPGA signals when a sampling or display key is pressed, receives **ADC data via SPI**, and measures the **power supply voltage using its internal ADC**. The processed data is then displayed on an **OLED screen via I2C**. To achieve precise control and efficient data handling, **both SPI and ADC are configured at the register level**, as detailed in the following sections.





SPI pins on Arduino

7.1. SPI configuration on Arduino

Based on datasheet for configuration on register level for SPI configuration as blow need to be implemented:

21.2 Register Description

21.2.1 SPCR – SPI Control Register

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – SPIE: SPI Interrupt Enable**

This bit causes the SPI interrupt to be executed if SPIF bit in the SPSR Register is set and the if the Global Interrupt Enable bit in SREG is set.

- **Bit 6 – SPE: SPI Enable**

When the SPE bit is written to one, the SPI is enabled. This bit must be set to enable any SPI operations.

- **Bit 5 – DORD: Data Order**

When the DORD bit is written to one, the LSB of the data word is transmitted first.

When the DORD bit is written to zero, the MSB of the data word is transmitted first.

- **Bit 4 – MSTR: Master/Slave Select**

This bit selects Master SPI mode when written to one, and Slave SPI mode when written logic zero. If $\overline{SS}$ is configured as an input and is driven low while MSTR is set, MSTR will be cleared, and SPIF in SPSR will become set. The user will then have to set MSTR to re-enable SPI Master mode.

- **Bit 3 – CPOL: Clock Polarity**

When this bit is written to one, SCK is high when idle. When CPOL is written to zero, SCK is low when idle. Refer to Figure 21-3 on page 196 and Figure 21-4 on page 196 for an example. The CPOL functionality is summarized in Table 21-3.

Table 21-3. CPOL Functionality

CPOL	Leading Edge	Trailing Edge
0	Rising	Falling
1	Falling	Rising

- **Bit 2 – CPHA: Clock Phase**

The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK. Refer to Figure 21-3 on page 196 and Figure 21-4 on page 196 for an example. The CPOL functionality is summarized in Table 21-4.

Table 21-4. CPHA Functionality

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample

- **Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0**

These two bits control the SCK rate of the device configured as a Master. SPR1 and SPR0 have no effect on the Slave. The relationship between SCK and the Oscillator Clock frequency f_{osc} is shown in Table 21-5.

Table 21-5. Relationship Between SCK and the Oscillator Frequency

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

Based on the datasheet, **SPCR – SPI Control Register** bits 7 and 6 need not be set for enabling and interrupt, while bits 5, 4, 3, 2, and 1 need to be zero for MSB first, slave mode, CPOL 0, CPHA 0, and a clock of 4 MHz (16 MHz micro clock / 4) respectively. These configurations are implemented in the Arduino code as follows:

```
// Function to initialize SPI in slave mode in register level
void initSPI() {
  DDRB &= ~(1 << DDB4);     // Set PB4 (SPI MISO) output
        //make these bits 1
  SPCR = (1 << 7)           // Enable SPI SPCR (SPIE=1)
       | (1 << 6);          // Enable SPI (SPE=1)
        //make these bits 0
  SPCR &= ~((1<<5)          //MSB first (DORD=0)
       | (1<<MSTR)          //SLAVE (MSTR=0) = (1<<4)
       | (1<<3)             //CPOL=0
       | (1<<2)             //CPHA=0
       | (1<<SPR1)          //SPI CLK (SPR1,SPR0=0)
       | (1<<SPR0));        //SPI CLK=16Mhz/4=4Mhz
}
```

7.2. ADC configuration on Arduino

The **Arduino Mega 2560's ADC** converts analog power supply measurements into digital values for processing and display. While it operates at **10-bit resolution**, scale mapping to **8-bit** is performed in the code to ensure better comparison with the **external ADC connected to the FPGA**. Based on datasheet for configuration on register level for ADC configuration as blow need to be implemented:

26.8 Register Description

26.8.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 26-3. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

Table 26-3. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection[1]
0	0	AREF, Internal V_{REF} turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Internal 1.1V Voltage Reference with external capacitor at AREF pin
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Note: 1. If 10x or 200x gain is selected, only 2.56V should be used as Internal Voltage Reference. For differential conversion, only 1.1V cannot be used as internal voltage reference.

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC Data Register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC Data Register immediately, regardless of any ongoing conversions. For a complete description of this bit, see "ADCL and ADCH – The ADC Data Register" on page 286.

- **Bits 4:0 – MUX4:0: Analog Channel and Gain Selection Bits**

The value of these bits selects which combination of analog inputs are connected to the ADC. See Table 26-4 for details. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set).

Table 26-4. Input Channel Selections

MUX5:0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
000000	ADC0	N/A		
000001	ADC1			
000010	ADC2			
000011	ADC3			
000100	ADC4			
000101	ADC5			
000110	ADC6			
000111	ADC7			

26.8.3 ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – ADEN: ADC Enable**

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**

In Single Conversion mode, write this bit to one to start each conversion. In Free Running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

- **Bit 5 – ADATE: ADC Auto Trigger Enable**

When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in ADCSRB.

- **Bit 4 – ADIF: ADC Interrupt Flag**

This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag. Beware that if doing a Read-Modify-Write on ADCSRA, a pending interrupt can be disabled. This also applies if the SBI and CBI instructions are used.

- **Bit 3 – ADIE: ADC Interrupt Enable**

When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.

- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

These bits determine the division factor between the XTAL frequency and the input clock to the ADC.

Table 26-5. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Based on the datasheet for **ADMUX – ADC Multiplexer Selection Register**, if bits 7 and 6 are set to 01, use AVCC with an external capacitor at the AREF pin as the reference voltage for the ADC. While bit 5 is set to zero to right-adjust the ADC result (LSB to the right), the 3 least significant bits of ADMUX must be set to zero to select ADC channel 0 (A0 in this case)

Also, in **ADCSRA – ADC Control and Status Register** A, bit 7 (ADEN) needs to be set to enable the ADC, and bit 2 (ADPS2) needs to be set to configure the prescaler to 16 (for a 16 MHz CPU, resulting in a 1 MHz ADC clock). These configurations are implemented in the Arduino code as follows:

```
// Function to initialize the ADC in register level
void initADC() {
  ADMUX = (1 << REFS0);          // 6/Set reference voltage to AVCC (physical pin on micro, connected to VCC)
  ADMUX &= ~(1 << ADLAR);        // 5/Right adjust the ADC result/LSB form right or left write
  ADMUX |= (0 & 0x07);           // make 3 LSBits zero / Select ADC channel (A0 here) p282 datasheet

  ADCSRA = (1 << ADEN)           // Enable ADC
         | (1 << ADPS2)          //2th bit need to be 1/ Set prescaler to 16 (for 16 MHz CPU, gives 1 MHz ADC clock);
} //16Mhz CPU clock, for ADC need to reduce this F with prescaler p285
```

7.3. OLED Display (0.96-Inch)

The **0.96-inch dual-color OLED display** from **Univision Technology** features a **128x64 pixel resolution** and supports **blue and yellow** colors. It uses the **SSD1306 driver IC**, enabling easy communication via the **I2C protocol** with just two pins (**SCL** and **SDA**). Operating at a voltage range of **2.5V to 5V** with a low current consumption of **20mA**, this display offers **high contrast**, **wide viewing angles** (over **160 degrees**), and **low power usage** (~0.04W). Unlike LCDs, OLEDs are **self-illuminating**, eliminating the need for a backlight, resulting in a **thinner, lighter** design with **sharper visuals**.

Pin Configuration:

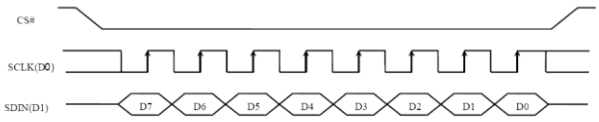
VCC Power Supply (3.3V–5V)

GND Ground

SCL Serial Clock (I2C)

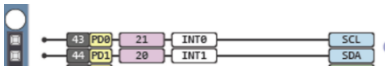
SDA Serial Data (I2C)

While its power section is supplied, the **0.96-inch OLED display** uses **I2C parallel communication**, where data and clock signals are sent through **two wires (SDA and SCL)**, with the **SSD1306 driver** interpreting commands and data to control pixel display.



These pins are connected respectively to Arduino, and since in Arduino we use “Wire.h” for I2C connection, they are synchronized automatically.

The following section described well commented Arduino code:




```
#include <avr/interrupt.h>  // Include for AVR interrupt handling
#include <Wire.h> // I2C library for OLED communication
#include <Adafruit_GFX.h> // Graphics library for OLED
#include <Adafruit_SSD1306.h> // OLED display driver

// Define OLED display parameters
#define SCREEN_WIDTH 128  // OLED display width in pixels
#define SCREEN_HEIGHT 64  // OLED display height in pixels
#define OLED_RESET     -1 // No reset pin used (-1 for sharing Arduino reset)
#define SCREEN_ADDRESS 0x3C // OLED I2C address (0x3D for 128x64, 0x3C for 128x32)

// Initialize OLED display object
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

// Global variables for SPI and ADC data handling
byte data; // Variable to store the received byte from SPI (8-bit)
byte adcarray[8]; // Array to store sampled ADC values
int indexW = 0;   // Write index for ADC data storage
int indexR = 0;   // Read index for retrieving stored ADC values

void setup() {   // Runs only once on startup
  Serial.begin(115200);       // Initialize serial communication
  Serial.println("Start up!!!"); // Startup message for debugging
  pinMode(SS, INPUT_PULLUP); // Configure SPI Chip Select (SS) as an input with pull-up resistor

  // Initialize OLED display
  if (!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("SSD1306 allocation failed")); // Display initialization failure message
    for (;;); // Halt execution if OLED fails
  }

  initSPI();  // Initialize SPI in slave mode (register level configuration)
  initADC();  // Initialize ADC in register level
  sei();      // Enable global interrupts for SPI communication

  // Display initial startup message
  display.clearDisplay();
```

```
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.println(F("Start up!!!"));
  display.display();
}

void loop() {    // Continuously runs (similar to while loop)
  uint16_t adcvalue = readADC(); // Read internal ADC value in real-time

  // Update OLED display
  display.clearDisplay();
  display.setCursor(0, 0);
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);

  display.print(F("ADC REAL-TIME: ")); // Display ADC label
  display.print(adcvalue);  // Show real-time ADC value

  display.setCursor(0, 10);         //
  display.print(F("SMPL: "));    //
  display.print(indexW-1);              //

  display.setCursor(60, 10);         //
  display.print(F("DIS: "));    //
  display.print(indexR-1);              //

  display.setCursor(0, 30); // Move cursor for FPGA ADC display
  display.print(F("FPGA ADC ----> "));
  display.print(data); // Show last received SPI data from FPGA

  display.setCursor(0, 40); // Move cursor for Arduino ADC display
  display.print(F("Arduino ADC -> "));
  display.print(adcarray[indexR]); // Show stored ADC value

  display.display();
  delay(300); // Delay for readability

  // Optional: Implement averaging for better display output (DIY)
}

// Function to initialize SPI in slave mode at the register level
void initSPI() {
  DDRB &= ~(1 << DDB4);  // Set MISO (PB4) as an output

  // Configure SPI Control Register (SPCR)
  SPCR = (1 << 7)  // Enable SPI interrupt (SPIE=1)
       | (1 << 6); // Enable SPI module (SPE=1)

  // Configure SPI settings (clearing necessary bits)
  SPCR &= ~((1 << 5)  // MSB first (DORD=0)
          | (1 << MSTR)  // Set as SPI slave (MSTR=0)
          | (1 << 3)  // Clock polarity CPOL=0
          | (1 << 2)  // Clock phase CPHA=0
          | (1 << SPR1)  // Set SPI clock speed (SPR1, SPR0=0)
          | (1 << SPR0)); // SPI clock speed = 16MHz / 4 = 4MHz
}

// SPI Interrupt Service Routine (ISR) - Handles SPI data reception
ISR(SPI_STC_vect) {
  data = SPDR;  // Read received data from SPI Data Register (SPDR) and store in variable

  if (data == 115) { // If received value is 115 ('s' in ASCII), indicates sampling key pressed
    int value = readADC();  // Read internal ADC value
    value = map(value, 0, 1023, 0, 255); // Scale 10-bit ADC result to 8-bit
    // Cannot use bit-shifting (<<) since MSB would be lost
    adcarray[indexW] = value; // Store scaled ADC value in array

    Serial.print("IndexW ");
    Serial.println(indexW); // Print write index for debugging
    indexW++;
    if (indexW == 8) indexW = 0; // Reset index after storing 8 samples
  }
  else { // If received data is not 's', it is an ADC value sent from FPGA
    Serial.print("IndexR ");
    Serial.println(indexR); // Print read index for debugging
    Serial.print("FPGA ADC: ");
    Serial.println(data); // Print received FPGA ADC value
    Serial.print("Arduino ADC: ");
    Serial.println(adcarray[indexR]); // Print stored Arduino ADC value
    Serial.println("##################################"); // Debug separator
    indexR++;
    if (indexR == 8) indexR = 0; // Reset index after reading 8 samples
  }
}

// Function to initialize the ADC at the register level
// Must reset Arduino each time FPGA resets
void initADC() {
  ADMUX = (1 << REFS0);          // Set reference voltage to AVCC (connected to VCC)
  ADMUX &= ~(1 << ADLAR);        // Right-adjust ADC result (LSB format)
  ADMUX |= (0 & 0x07);           // Select ADC channel (A0) - Datasheet reference: p282

  ADCSRA = (1 << ADEN)           // Enable ADC module
```

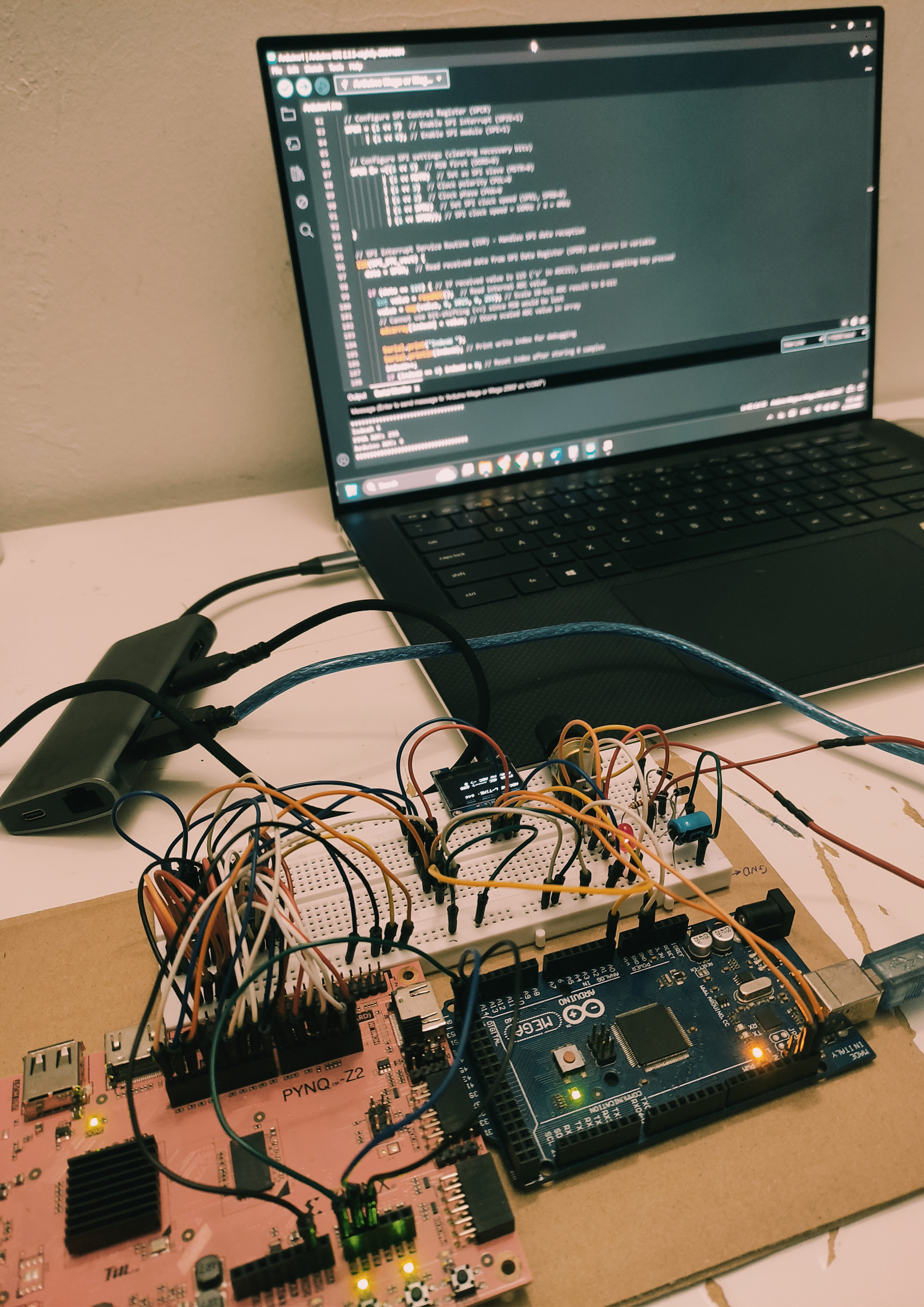
```
        | (1 << ADPS2);         // Set prescaler to 16 (16MHz / 16 = 1MHz ADC clock) - Datasheet reference: p285
}

// Function to read ADC value without using analogRead() (manual ADC conversion)
uint16_t readADC() {
  ADCSRA |= (1 << ADSC);         // Start ADC conversion (single conversion mode)
  while (ADCSRA & (1 << ADSC));  // Wait until conversion is complete (bit clears when done)
  return ADC;                    // Return the ADC result
}
}
```

Conclusion

This project successfully implements a **digital voltmeter system** by integrating an **FPGA-based ADC** and a **microcontroller ADC**, demonstrating the differences in precision, speed, and memory management between the two approaches. The **Arduino Mega 2560** was configured at the **register level** for SPI and ADC, enabling direct communication with the FPGA, which handled an **external ADC (ADC0831N) and SRAM (HY6116ALP)** for efficient data storage. The system effectively measured the voltage from a **linear power supply**, displayed results on an **OLED screen via I2C**, and utilized **SPI communication** for real-time data transfer. By comparing internal and external ADC performance, this project provides a deeper understanding of **embedded systems design, FPGA interfacing, and digital measurement techniques**. The knowledge gained from this work can be extended to **industrial monitoring systems, sensor data acquisition, and real-time digital signal processing applications**.

References

1. **Electronics for Embedded Systems**, Prof. Claudio Passerone, Politecnico di Torino
2. **Rohde & Schwarz Teaching Materials** – Fundamentals of Embedded Systems
3. Altera (Intel) DE1 Development Board Documentation
4. Arduino Mega 2560 Datasheet and Reference Manual
5. Texas Instruments – **Introduction to SPI and I2C Communication**
6. Microchip Technology – **Analog-to-Digital Conversion Guide**
7. Online resources: **YouTube tutorials, embedded systems forums, and technical blogs on FPGA and ADC design**


PYNQ-Z2
ARDUINO
MEGA