

**School of**

**Electronics and Communication Engineering**

**Minor Project Report**

**on**

**RISC V Based SoC Design-SPI**

**By:**

1.

**Rahul G Teli**

USN: 01FE21BEE008

2.

**Malhar Kulkarni**

USN: 01FE21BEE016

3.

**Chandru Thomare**

USN: 01FE21BEE022

4.

**Chandrashekhar Angadi**

USN: 01FE21BEE031

**Semester: V, 2023-2024**

Under the Guidance of

**Dr.Saroja Siddamal**

|  |
| --- |
| **K.L.E SOCIETY’S**  **KLE Technological University, HUBBALLI-580031 2022-2023**    SCHOOL OF ELECTRONICS AND COMMUNICATION ENGINEERING  **CERTIFICATE**  This is to certify that project entitled **“RISC V Based SoC Design-SPI”** is a bonafide work carried out by the student team of **” Rahul G Teli- 01FE21BEE008, Malhar Kulkarni- 01FE21BEE016, Chandru Thomare - 01FE21BEE022, Chandrashekar R Angadi - 01FE21BEE031 ”**. The project report has been approved as it satisfies the requirements with respect to the minor project work prescribed by the university curriculum for BE (V Semester) in School of Electronics and Communication Engineering of KLE  Technological University for the academic year 2022-2023.  **Dr. Saroja Siddamal Dr. Nalini C Iyer Dr. Basavaraj S Anami**  **Guide Head of School Registrar**  **External Viva:**  **Name of Examiners Signature with date**  1.  2. |
| **ACKNOWLEDGMENT**  We would like to express our sincere gratitude to KLE Technological University, Hubballi for providing us the facilities and resources to carry out our project on RISC-V Based SoC Design with SPI interface. We are especially thankful to the Head of School of Electronics and Communication, Dr. Nalini C. lyer, for her constant motivation and guidance throughout this endeavour. We would also like to thank our project guide Dr. Saroja V. Siddamal for her invaluable insights, feedback, and direction at every stage of this project. Her vast knowledge and experience in this field steered us in the right direction whenever we faced any difficulties. In addition, we are grateful to the entire VLSI Design faculty team for clarifying our doubts patiently and helping us gain a thorough understanding of advanced concepts related to our project. We also appreciate the lab technicians for promptly addressing all our queries on the software tools and ensuring smooth progress of simulations. Finally, we would like to express our heartfelt gratitude towards our parents and family members for their unconditional love, emotional support and encouragement that gave us the motivation to work diligently on this challenging yet fulfilling project. We dedicate our project work to them as without their blessings, none of this would have been possible.Once again, we thank everyone who was directly or indirectly involved in making our efforts come to fruition. This acknowledgment reaffirms our sincere appreciation  - Rahul G Teli, Malhar Kulkarni, Chandru Thomare, Chandrashekar R Angadi |
| **ABSTRACT**  As embedded systems grow more complex, effective communication between components becomes critical for overall system reliability and efficiency. The serial peripheral interface (SPI) has emerged as a popular bus protocol for achieving high-speed synchronized data transfer with reduced pin requirements. This project focuses on designing and integrating an SPI controller subsystem into a RISC-V based System-on-Chip (SoC). The SPI controller is implemented in Verilog HDL and seamlessly interfaces with the RISC-V processor core to enable communication with external peripherals. It supports essential features including full-duplex data transmission, programmable clock polarity and phase options, interrupt generation and arbitration logic for multi-master configurations. A detailed register set with configurable control bits and data buffers allows flexibility to connect to a wide range of SPI-based sensors, ADCs, displays as per application requirements. Comprehensive testing procedures in simulation validate error-free functionality for corner cases. The design is mapped to the ARTY-7 FPGA development board for prototype demonstration. The realized SoC allows the RISC-V processor to efficiently orchestrate data exchange with multiple external devices using the integrated SPI subsystem. This simplifies integration challenges in embedded product designs and serves as a vital building block for realizing more complex SoC architectures. The modular Verilog implementation, register specifications and test methodology documented in this project will facilitate future enhancements as well as integration of the design into variants of RISC-V based SoCs |

|  |  |  |
| --- | --- | --- |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  |  |  |
|  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  |  |  | |
| **Chapter 1**  **Introduction**  The Serial Peripheral Interface (SPI), a protocol designed for the efficient, synchronous communication between devices, is fundamental in the realm of embedded systems. Originating in the mid-1980s by Motorola for inter-chip communication, SPI has become a staple in facilitating dialogue between various slow-speed devices, including sensors, analog-to-digital converters, real-time clocks, and memory modules like flash memory. This protocol, characterized by its full-duplex communication ability, leverages a select line alongside separate clock and data connections to manage multiple devices, thereby providing an uninterrupted data transfer capability that supports up to 32 bits of simultaneous transmission  **1.1 Motivation**  The urgency to monitor environmental quality, characterized by the analysis of gases and impurities both indoors and outdoors, necessitates innovative technological solutions. Traditional systems for detecting air pollution levels are often hampered by their lack of mobility and prohibitive costs. Herein lies the potential of System on Chip (SoC) technology, specifically those powered by the RISC V processor integrated with SPI, I2C, UART, and GPIO for sensor-based applications. SPI's full-duplex mode, alongside its capability for 32-bit data transfer and minimal power consumption due to its straightforward hardware configuration, underscores its advantage over alternatives like I2C.  **1.2 Objectives**  The primary goals of this endeavour are twofold:  To seamlessly integrate SPI protocol with the RISC V ET1032 Processor.  To conduct conclusive testing, verification, and validation of the SPI protocol within a RISC V based SoC, using the Arty-A7 FPGA board as a platform. | | |
| **1.3 Literature survey**  **NXP-SPI Block Guide V03.06 by Motorola,Inc.**  The overview, features, and modes of operation are covered in the introduction.The master and  slave wiring connection, which consists of MOSI (Master Out Slave In), MISO (Master In Slave  Out), SCK (Line for Clock Signal), and SS/CS (Slave Select/Chip Select), is described in the  external signal description. SPI Control Register- 1, SPI Control Register- 2, SPI Baud Rate  Register, SPI Status Register, and SPI Data Register are among the registers described in the  memory map. Master and Slave modes, transmission formats, clock phase controls, and polarity  controls are all included in the functional description. It includes the CPHA=0 and CPHA=1  transfer formats, SPI baud rate generation, slave select output. [4].  **Design and Implementation of a High-Speed Serial Peripheral Interface**  A synchronous protocol called Serial Peripheral Interface enables serial communication between  a slave and a master device. This work aims to give a detailed account of an implementation of  a high speed SPI Master/Slave. The designs are based on SPI Block Guide V03.06 by Motorola.  The design phases are methodically developed, beginning with the preliminary specifications  and ending with the final physical design. The entire design is mapped into Xilinx’s Virtex 5  FPGA chips using Verilog 2001. A prominent serial protocol for low- to medium-speed data  stream transfers within and between chips is serial peripheral interface (SPI). It facilitates communication  between a microcontroller and several additional devices, such as external DACs,  ADCs, and EEPROMs [1].  **IP Core of Serial Peripheral Interface(SPI) with AMBA APB Interface**  In this paper, the SPI design has been successful in obtaining a maximum 16 MHz frequency  and a complete duplex 8 bit serial data transfer that is solely for master mode. MODELSIM  is used to model the job, while QUARTUS lite 16 is used to synthesis it. The RTL netlist  viewer from QUARTUS light 16’s findings demonstrate that data is transported from one submodule to another submodule. This design is a straightforward interface that connects easily to microcontroller’s (I/O) port, and it can be used to connect to an APB [5].  **Implementation of SPI Protocol in FPGA**  This paper describes the implementation of the Serial Peripheral Interface (SPI) protocol on  a Field Programmable Gate Array (FPGA). State machine diagrams are used in the implementation of the SPI Master and SPI Slave components. Very High Speed Integrated Circuits  Hardware Descriptive Language (VHDL) is used for the coding. After obtaining an acknowledgment, the received data is examined and the simulated data is displayed [7].  **Design and Implementation of Serial Peripheral Interface Protocol Using Verilog**  HDLThis study aims to design and build a master and slave SPI (serial peripheral interface) using  Verilog HDL. One kind of serial communication protocol that transports synchronous serial data  in full duplex mode is the serial peripheral interface, or SPI. In SPI, there are two different communication modes: master and slave. The Xilinx ISE design suite 13.3.1 is used to simulate and synthesize the entire design while the master device creates a serial clock and the slave devices are permitted with individual slave select lines. A hardware/firmware communications protocol called Serial to Peripheral Interface (SPI) was created by Motorola and eventually embraced by other companies in the sector. SPI is also referred to as a ”four wire” serial bus occasionally. Many microprocessor/microcontroller peripheral chips employ the Serial Peripheral Interface, or SPI-bus, a straightforward 4-wire serial communications interface that allows controllers and peripheral devices to connect with one another. SPI is primarily designed to facilitate communication between a host CPU and peripherals, but it can also be used to connect two processors.  When the SPI bus is fully operational, it is a synchronous data link configuration featuring a  Master/Slave interface. SPI allows for the possibility of both single-master and multi-master  protocols. Typically, only the PCB uses the SPI Bus. The SPI Bus was created to transport  data quickly and efficiently between different IC chips [6].  **Design and Analysis of Serial Peripheral Interface for Automotive Controller**  A popular communication protocol called Serial Peripheral Interface (SPI) enables serial data  transfer across a small distance between a slave and a master device. However, there are issues  with clock synchronization and poor speed in the existing Serial Peripheral Interface implementation.  A few factors related to automotive must be taken into account in order to improve the  interface device’s performance. The design, simulation, verification, and optimization of SPI  based on automotive interface device standards are presented in this work. It includes improved  power dissipation efficiency and a faster interface device. The speed and area attained are far  better than the SPI’s present design. Because of the technical relationship between power and  speed in VLSI design, even though the total dissipation obtained is more than the already used  SPI architecture, it is acceptable [2].  **Design and Simulation of SPI Master / Slave Using Verilog HDL**  This paper’s goal is to use Verilog HDL to develop and simulate an SPI (serial peripheral interface)  master and slave. It is a widely used interface for attaching peripherals to microprocessors  and to each other. Although various Motorola microcontrollers support transfers of any range  of blocks between two and sixteen bits at a time, the majority of literature claims that the  interface can only be utilized for eight or sixteen bit block data transfers. Control signals can  be used to implement data transfers of more than sixteen bits at a time with ease due to the  serial nature of the interface. Either the slave or the master protocol can be implemented by  the SPI. Up to 32 separate SPI slaves can be under the control of the SPI when it is set as a  master. The transmit and receive register widths can be adjusted to be wider than sixteen bits.  As many devices as there are pins on the core microcontroller can be connected using SPI. The  communication rate amongst ICs is substantially higher. With SPI, full duplex communication  is carried out [8].  **Design and Verification Serial Peripheral Interface (SPI) Protocol for Low Power Applications**  High-speed data communication between devices is made possible by Serial to Peripheral Interface (SPI) technology, which was intended to replace parallel connections and eliminate the  need to route parallel buses around PCBs. When connecting its initial 68000-based microcontroller unit to peripheral functions in the late 1970s, Motorola was the first firm to identify the circuit technology SPI, which was eventually adopted by other companies in the industry. SPI gained popularity as a communication protocol due to its ease of interface and speed, which facilitates data transfers with ease. SPI has established a strong position for itself in embedded systems, whether they be system-on-a-chip (SOC) processors or other microcontrollers like PIC and AVR, as well as more sophisticated 32-bit processors like those that use MIC, Power PC, or ARM. These chips typically come with SPI controllers, which can operate in either slave or master mode. SPI is occasionally used for communication in chip-based or field programmable gate array devices. SPI is therefore a widely utilized technology nowadays for communicating with peripheral devices when fast data transfer under real-time limitations is required. 10  Intel is actively working on the Enhanced Serial Peripheral Interface bus (eSPI), a replacement  for their low pin count (LPC) bus.In contrast to systems using low pin counts, enhanced  SPI was designed to allow greater throughput than LPC, lower working voltages to 1.8 volts to  enable smaller chip manufacturing processes, and allow enhanced SPI to share serial peripheral  interface flash devices with the host. On the other hand, low pin counts (LPC) buses prevented  LPC peripherals from using firmware hubs. Enhanced serial peripheral interface also allows  system designers to trade off cost and performance. The extended serial peripheral interface bus  can be separated from the SPI bus to enable higher performance, or it can be shared with SPI  devices to conserve pins [3].  A diversified body of literature underscores the importance and versatility of the SPI protocol in embedding communication systems. From Motorola's foundational NXP-SPI Block Guide that delineates the operational mechanics and architectural specifics of SPI, to contemporary papers discussing high-speed SPI master/slave implementation on FPGA chips using Verilog, the spectrum of research showcases SPI's pivotal role in enhancing microcontroller communication frameworks. Special emphasis on its employment in automotive controller design, wherein clock synchronization and speed are critical, further exemplify SPI's adaptability to rigorous performance standards. Moreover, the advent of the Enhanced Serial Peripheral Interface bus (eSPI) by Intel demonstrates a forward-looking approach to surpassing traditional limitations of serial communication in favor of more efficient, scalable solutions. | | |
| **1.4 Problem statement**  This project aims not only at the design and implementation of the SPI protocol but also at its integration with the RISC V processing architecture. This is supplemented by a rigorous phase of testing and validation to ascertain the efficacy of the implemented design  **1.5 Application in Societal Context**  The application spectrum of the SPI protocol significantly intersects with societal benefit, particularly in enhancing the functionality and interactivity of digital storage (MMC and SD cards), conversion modules (ADCs and DACs), and communication interfaces (Ethernet, USB, USART). Its utilization in real-time monitoring devices such as temperature and pressure sensors, as well as in consumer electronics like touchscreens and camera lenses, illustrates SPI's omnipresence in modern technology infrastructure. This ubiquity, indicative of SPI's reliability and efficiency, solidifies its position as an instrumental asset in the development of responsive and adaptable embedded systems.  **1.6 Organization of report**  • Chapter 1: Introduction  It includes motivation towards the project, objectives of project, literature survey done towards  the problem statement, defining the problem statement and looking at the application in a  societal context, followed by project planning.  • Chapter 2: System design This chapter includes the High level functional block diagram and description of the functional blocks.  • Chapter 3: Implementation details  This chapter contains detailed information about implementation. It also Includes specifications  used to design and propose the final system architecture. It contains the Hardware Interfacing  Description with in-depth details in Firmware, Hardware Description and the Port Mapping. It  also includes the detailed information about the Registers Description and Finite State Machine.  • Chapter 4: Results and Discussions  This chapter includes the conclusion and results of implementation.  • Chapter 5: Conclusion  This is the concluding chapter that includes project closure and epilogue along with- the future scope of our project. | | |



**Chapter 2**

**System Design of SPI Integration**

This chapter elucidates the block-level design of the Serial Peripheral Interface (SPI) within a System on Chip (SoC) architecture. The focal point of this design is the seamless integration of the Processor ET1032 with the SPI controller and SPI master to facilitate efficient data transfer to the SPI slave. The architecture underpins the transition of data handling from parallel to serial transmission, underscoring the system’s capability to bridge the computational core with peripheral communication modules effectively.

**2.1**

**Overview of the System ::Architecture:**

Figure 2.1:

Block Diagram of SPI

**ET1032-Processor**

The Processor consists of boot memory from where it fetches the instruction pointed by the program counter(PC), decodes that instruction and gives values for different variables based on the instruction. addrb and dinb are the main signals that are configured. These Processor variables are connected with the SPI Controller block.

**SPI Controller**

SPI Controller receives values for PADDR and PWDATA from the Processor. Depending on the PADDR values, the two control registers and the baud rate registers are configured, thereby the read or write operations are decided. This block provides different values of frequencies of clock as input to the Master block based on the SPIBDR register bits.

**SPI Master**

The signals SCLK, MOSI and SS are the output from the master and MISO is the input to the master. The data register consists of input write data register, which is transferred to the slave serially through the MOSI line based on SCLK and the read data register receives the input through the MISO line based on SCLK.

**Data Transfer Mechanism**

Data transfer within the system initiates with the Processor ET1032 transmitting data in a parallel format to the SPI Controller, which, in turn, processes and transmits this data to the SPI Master while maintaining the parallel configuration. The SPI Master then converts this parallel data into a serial format, facilitating the transmission to the SPI slave. This structured approach to data handling, transitioning from parallel to serial transmission, epitomizes the system’s design philosophy, optimizing for speed and efficiency within the embedded SoC environment.

The system's architecture is predicated on a tripartite structure comprising the Processor ET1032, the SPI Controller, and the SPI Master. At the outset, data is orchestrated in a parallel format from the Processor ET1032 to the SPI Controller, ensuring high-speed data handling within the core system. Subsequently, the SPI Controller interfaces with the SPI Master, maintaining the parallel data framework to uphold the integrity and speed of data processing. The culmination of this data journey is marked by the transition from parallel to serial transmission as the SPI Master communicates with the SPI slave, signifying the core-to-peripheral data exchange pivotal to SPI’s operational paradigm.

|  |
| --- |
| **Chapter 3**  **Implementation details**  **3.1 SPI Protocol Architecture**  This chapter delves into the intricate details of implementing the Serial Peripheral Interface (SPI) protocol. It encompasses a comprehensive examination of the system's architecture, including the finite state machine (FSM) of the SPI Master, and culminates in the elucidation of the final system’s architecture. The dual nature of the implementation—firmware and hardware—is meticulously explored, offering insights into the functional and control mechanisms underlying the SPI protocol within the System on Chip (SoC) framework..  **3.2 SoC Integration Design**    The architecture of the SPI protocol is fundamentally designed to streamline the communication between the processor and peripheral devices. Figure 3.1 illustrates the structural overview, featuring both the Processor and the SPI top module. The interplay begins with the Processor dispatching clk\_p, reset, and addrb signals to the SPI top module, which subsequently configures its registers accordingly. This configuration precipitates the serial data transfer from the master to the slave via the Master Out Slave In (MOSI) line, ensuring a seamless flow of information.  3.2.1 Firmware Implementation  Implemented within an Ubuntu environment, the firmware comprises four primary files: SPI.c, main.c, config.h, and SPI.h, collectively underpinning the SPI’s operation.  SPI.c houses the core SPI program logic, mapping considerable functionality into the SPI operations. main.c is responsible for invoking functions defined within SPI.c, effectively serving as the entry point of the firmware. config.h specifies system-wide parameters, including the base address of the SPI (0x30000600). SPI.h declares essential variables correlating to SPI Control Register (SPI\_CR), SPI Transmit Data Register (SPI\_PWDATA), SPI Receive Data Register (SPI\_PRDATA), and the SPI Status Register (SPI\_SR), with designated addresses for efficient data handling and operational control.  Compilation is streamlined via a simple command (./build.sh) executed within the terminal, culminating in the generation of a soc\_32.mif file containing binary instructions for the processor’s boot memory.    **3.2.2 Hardware Synthesis**  The SoC hardware framework incorporates the RISC-V Processor, SPI Peripheral, memory locations, and a Clock Wizard for internal synchronization, illustrated in Figure 3.3. The Clock Wizard regulates instruction execution speeds, while the processor’s modular architecture facilitates a wide spectrum of applications, from embedded devices to high-performance computing. The parallel existence of two memory blocks augments the system’s capacity to manage data and instructions efficiently.    **3.2.3 Port Mapping Considerations**  Port mapping, illustrated in Figure 3.4, signifies the logical connection between the processor’s control and data signals to those of the SPI, enabling coherent communication within the SoC.    **3.3 Register Configurations and Descriptions**  Integral to the SPI's functionality are several key registers: Control Register 1 (SPICR 1), Control Register 2 (SPICR 2), the Status Register (SPISR), and the Baud Rate Register.  SPICR 1 facilitates primary control operations, such as SPI enable/disable, master/slave mode selection, and clock polarity/phase settings.  SPICR 2 primarily deals with serial pin control distinctions between master and slave modes.  SPISR indicates transaction statuses, including interrupt flag, transmit empty flag, and mode fault occurrences.  The Baud Rate Register governs the SPI communication speed, adjustable through specific preselection and selection bits to match system requirements.  **Register Descriptions**   1. **SPI Control Register 1 [8 bit] [Read/Write]:**      * SPIE — SPI Interrupt Enable Bit   This bit enables SPI interrupt requests, if SPIF or MODF status flag is set.  1 = SPI interrupts enabled.  0 = SPI interrupts disabled.  the functions of the SPIE bit:   * Enable interrupts for SPI transmission: When SPIE is set to 1 and the TXIE bit is set to 1, an interrupt is generated when the SPI transmission is complete. * Enable interrupts for SPI reception: When SPIE is set to 1 and the RXIE bit is set to 1, an interrupt is generated when a byte of data is received from the slave device. * Disable interrupts for SPI: When SPIE is set to 0, all SPI interrupts are disabled. * SPE — SPI System Enable Bit   This bit enables the SPI system and dedicates the SPI port pins to SPI system  functions.  1 = SPI enable, port pins are dedicated to SPI functions.  0 = SPI disabled (lower power consumption).  **FUNCTIONS:**   * Enable the SPI module: When SPE is set to 1, the SPI module is enabled and can be used to communicate with SPI devices. * Disable the SPI module: When SPE is set to 0, the SPI module is disabled and cannot be used to communicate with SPI devices. * SPTIE — SPI Transmit Interrupt Enable   This bit enables SPI interrupt requests, if SPTEF flag is set.  1 = SPTEF interrupt enabled 0 = SPTEF interrupt disabled.  **FUNCTIONS:**   * Enable interrupts for SPI transmissions: When SPTIE is set to 1, an interrupt is generated when the SPI transmission is complete. This interrupt can be used to notify the software that the SPI transmission is complete and that it can begin processing the transmitted data. * Disable interrupts for SPI transmissions: When SPTIE is set to 0, interrupts are disabled for SPI transmissions. This is useful when the software does not need to be notified when the SPI transmission is complete. * MSTR — SPI Master/Slave Mode Select Bit   This bit selects, if the SPI operates in master or slave mode. Switching the SPI from  master to slave or vice versa forces the SPI system into idle state.  1 = SPI is in Master mode.  0 = SPI is in Slave mode.  **FUNCTIONS:**   * Configure the SPI module as a master device: When MSTR is set to 1, the SPI module is configured as a master device. In this mode, the SPI module initiates SPI communications and controls the timing of SPI transfers. * Configure the SPI module as a slave device: When MSTR is set to 0, the SPI module is configured as a slave device. In this mode, the SPI module waits for SPI communications to be initiated by the master device and responds to the master's requests. * CPOL — SPI Clock Polarity Bit   This bit selects an inverted or non-inverted SPI clock. To transmit data between SPI  modules, the SPI modules must have identical CPOL values.  1 = Active-low clocks selected. In idle state SCK is high.  0 = Active-high clocks selected. In idle state SCK is low.  **FUNCTIONS:**   * Set the clock polarity to low when idle: When CPOL is set to 0, the clock signal is low when the SPI module is not transmitting or receiving data. This is the default polarity for SPI communication. * Set the clock polarity to high when idle: When CPOL is set to 1, the clock signal is high when the SPI module is not transmitting or receiving data. This polarity is sometimes used for SPI devices that require a high clock signal when they are not being used. * CPHA — SPI Clock Phase Bit   This bit is used to select the SPI clock format. In master mode, a change of this bit  will abort a transmission in progress and force the SPI system into idle state.  1 = Sampling of data occurs at even edges (2,4,6,...16) of the SCK clock  0 = Sampling of data occurs at odd edges (1,3,5,...,15) of the SCK clock  undefined  **FUNCTIONS:**   * Sample data on the first clock edge: When CPHA is set to 0, the data is sampled on the first clock edge. This is the default phase for SPI communication. * Sample data on the second clock edge: When CPHA is set to 1, the data is sampled on the second clock edge. This phase is sometimes used for SPI devices that require the data to be sampled on the second clock edge. * LSBFE — LSB-First Enable   This bit does not affect the position of the MSB and LSB in the data register. Reads  and writes of the data register always have the MSB in bit 7.  1 = Data is transferred least significant bit first.  0 = Data is transferred most significant bit first.  **FUNCTIONS:**   * Transfer data least significant bit first: When LSBFE is set to 1, data is transferred least significant bit first. This means that the least significant bit of the data is transferred first, followed by the next-least * significant bit, and so on. This is the default order for SPI communication. * Transfer data most significant bit first: When LSBFE is set to 0, data is transferred most significant bit first. This means that the most significant bit of the data is transferred first, followed by the next-most significant bit, and so on. This order is sometimes used for SPI devices that require the data to be transferred most significant bit first. * SSOE — Slave Select Output Enable   The SS output feature is enabled only in master mode, if MODFEN is set  **FUNCTIONS:**   * Enable the SS output: When SSOE is set to 1, the SPI module outputs the SS signal to the slave device. This signal is used to select the slave device that the master device wants to communicate with. * Disable the SS output: When SSOE is set to 0, the SPI module does not output the SS signal to the slave device. This means that the slave device is not selected, and it will not respond to SPI communication requests from the master device.  1. **SPI Control Register 2 [8 bit] [Read/Write]:**      * MODFEN — Mode Fault Enable Bit   This bit allows the MODF failure being detected. If the SPI is in Master mode and  MODFEN is cleared, then the SS port pin is not used by the SPI. In Slave mode, the SS is available only as an input regardless of the value of MODFEN.    1 = SS port pin with MODF feature  0 = SS port pin is not used by the SPI  **FUNCTIONS:**   * **Enable mode fault detection:** When MODFEN is set to 1, the SPI module will detect and report mode faults. A mode fault is a condition that occurs when the SPI module is configured incorrectly. For example, a mode fault can occur if the master and slave devices are configured with different clock polarities or phases. * **Disable mode fault detection:** When MODFEN is set to 0, the SPI module will not detect or report mode faults. This can be useful if you want to ignore mode faults and allow the SPI communication to continue even if there is a configuration error. * BIDIROE — Output enabled in the Bidirectional mode of operation   This bit controls the MOSI and MISO output buffer of the SPI, when in bidirectional  mode of operation (SPC0 is set). In master mode this bit controls the output buffer of  the MOSI port, in slave mode it controls the output buffer of the MISO port.  1 = Output buffer enabled  0 = Output buffer disabled  **FUNCTIONS:**   * Enable output buffers in bidirectional mode: When BIDIROE is set to 1, the output buffers of the MOSI and MISO ports are enabled. This allows the SPI module to output data on both the MOSI and MISO ports in bidirectional mode. * Disable output buffers in bidirectional mode: When BIDIROE is set to 0, the output buffers of the MOSI and MISO ports are disabled. This prevents the SPI module from outputting data on either the MOSI or MISO ports in bidirectional mode. * SPISWAI — SPI Stop in Wait Mode Bit   This bit is used for power conservation while in wait mode.  1 = Stop SPI clock generation when in wait mode.  0 = SPI clock operates normally in wait mode.  **FUNCTIONS:**   * Stop SPI clock in wait mode: When SPISWAI is set to 1, the SPI clock is stopped when the SPI module is in wait mode. This can help to reduce power consumption by preventing the SPI module from generating clock pulses when it is not being used. * Allow SPI clock to operate normally in wait mode: When SPISWAI is set to 0, the SPI clock operates normally even when the SPI module is in wait mode. This allows the SPI module to communicate with slave devices without any interruption, even if the microcontroller is in a low-power state. * SPC0 — Serial Pin Control Bit 0   **FUNCTIONS:**  **Bidirectional mode:** When SPC0 is set to 1, the SPI operates in bidirectional mode. In this mode, the MOSI pin is used as an input and the MISO pin is used as an output. This allows the SPI to be used for full-duplex communication, where both the master and slave devices can send and receive data simultaneously.  **Unidirectional mode:** When SPC0 is set to 0, the SPI operates in unidirectional mode. In this mode, the MOSI pin is used as an output and the MISO pin is used as an input. This allows the SPI to be used for half-duplex communication, where only one device can send data at a time.   1. **SPI Baud Rate Register [8 bit]:**       SPPR2–SPPR0 — SPI Baud Rate Preselection Bits  SPR2–SPR0 — SPI Baud Rate Selection Bits  These bits specify the SPI baud rates as shown in the table below. In master mode, a change of these bits will abort a transmission in progress and force the SPI system into idle state.   * + Functions of the SPPR2–SPPR0 bits: * Set the baud rate: The SPPR2–SPPR0 bits are used to set the baud rate of the SPI. The baud rate is calculated using the following formula:   Baud\_rate = f\_clk / (2 \* (SPPR2 + SPPR1 + SPPR0 + 1))   * Abort a transmission in progress: In master mode, a change of the SPPR2–SPPR0 bits will abort a transmission in progress and force the SPI system into idle state. This is because a change of the baud rate will affect the timing of the SPI communication, and it is not possible to change the baud rate in the middle of a transmission.   + **Applications of the SPPR2–SPPR0 bits:**   The SPPR2–SPPR0 bits are commonly used in applications that require a specific baud rate, such as:   * **Audio communication:** Audio devices often use SPI to transmit and receive audio data, and they require a specific baud rate for this communication. * **Sensor communication:** Sensors often use SPI to send data to a microcontroller for processing, and they require a specific baud rate for this communication. * **Actuator control:** Microcontrollers often use SPI to control actuators, such as motors and servos, and they require a specific baud rate for this communication.  1. **SPI Status Register [8 bit] [Only Read & Not Write]:**      * SPIF — SPIF Interrupt Flag   This bit is set after a received data byte has been transferred into the SPI Data  Register. This bit is cleared by reading the SPISR register (with SPIF set) followed by  a read access to the SPI Data Register.  1 = New data copied to SPIDR  0 = Transfer not yet complete  **Functions of the SPIF bit:**   * **Generate an interrupt when a new byte of data is received:** When SPIF is set to 1, an interrupt is generated when a new byte of data is received. This interrupt can be used to notify the software that a new byte of data is available and that it can begin processing the data. * **Disable interrupts for SPI reception:** When SPIF is set to 0, interrupts are disabled for SPI reception. This is useful when the software does not need to be notified when a new byte of data is received. * SPTEF — SPI Transmit Empty Interrupt Flag   If set, this bit indicates that the transmit data register is empty. To clear this bit and  place data into the transmit data register, SPISR has to be read with SPTEF=1,  followed by a write to SPIDR.  1 = SPI Data register empty.  0 = SPI Data register not empty.  **FUNCTIONS:**   * Check if the transmit data register is empty: The SPTEF bit can be used to check if the transmit data register is empty before writing new data to the register. This is important to ensure that the data is not lost. * Trigger an interrupt when the transmit data register is empty: The SPTEF bit can be used to trigger an interrupt when the transmit data register is empty. This can be used to notify the software that the SPI module is ready to accept new data. * Clear the transmit data register: The SPTEF bit can be cleared by reading the SPISR register with SPTEF=1, followed by a write to the SPIDR register. This will empty the transmit data register and prepare the SPI module to accept new data. * MODF — Mode Fault Flag   This bit is set if the SS input becomes low while the SPI is configured as a master and  mode fault detection is enabled, MODFEN bit of SPICR2 register is set. The flag is  cleared automatically by a read of the SPI Status Register (with MODF set) followed  by a write to the SPI Control Register 1.  1 = Mode fault has occurred.  0 = Mode fault has not occurred.  **FUNCTIONS:**   * Indicate that a mode fault has occurred: The MODF bit is set to 1 if a mode fault has occurred. This can be used to notify the software that there is a problem with the SPI communication and that corrective action needs to be taken. * Clear the MODF flag: The MODF flag can be cleared by reading the SPI Status Register (with MODF set) followed by a write to the SPI Control Register 1. This will reset the SPI module and clear the MODF flag. * Enable mode fault detection: The MODFEN bit in the SPICR2 register must be set to 1 in order for the SPI module to detect mode faults. If MODFEN is set to 0, then the SPI module will not detect mode faults and the MODF flag will never be set.  1. **SPI Data Register [8 bit]:**     The SPI Data Register is both the input and output register for SPI data. A write to  this register allows a data byte to be queued and transmitted. For a SPI configured as a  master, a queued data byte is transmitted immediately after the previous transmission  has completed.   * Received data in the SPIDR is valid when SPIF is set. * If SPIF is cleared and a byte has been received, the received byte is transferred from the receive shift register to the SPIDR and SPIF is set. * If SPIF is set and not serviced, and a second byte has been received, the second   received byte is kept as valid byte in the receive shift register until the start of another transmission. |
| **Chapter 4: Results and Discussions on SPI Protocol Verification**  This chapter elucidates the comprehensive verification of the SPI (Serial Peripheral Interface) protocol through simulated results and practical testing on an FPGA platform. It intricately details the outcome of RTL (Register-Transfer Level) coding written in Verilog HDL (Hardware Description Language), observed via the Vivado simulator, alongside hardware implementation results entailing testing the SPI peripheral with an Arduino as a slave and integrating an air quality monitoring system.  **4.1 Evaluation of Results**  The successful RTL coding for the SPI protocol demonstrates the feasibility and robust operation of the interface within a SoC (System on Chip) environment. Utilizing the Vivado simulator, waveforms for various operations, including write sequences, were meticulously examined.    **4.1.1 Insight into the SoC Top Module**  An overview of the SoC reveals a complex assembly hosting multiple protocols such as SPI, I2C, UART, and GPIO. Within this assemblage, SPI components are accentuated, underscoring their pivotal role in the system’s communication framework as illustrated in Figure 4.1.  **4.2 Simulation Outcomes and Analytical Overview**  A precise analysis of the SPI Write operation underscores the efficient transfer mechanism spearheaded by the SPI protocol. As delineated in Figure 4.2, data (PWDATA) extracted from the Processor is adeptly conveyed through the MOSI (Master Out Slave In) line in synchronization with the sclk (serial clock), emphasizing the MSB-first data transfer policy.    **4.3 Verification via Hardware Implementation**  The real-world efficacy of the SPI peripheral was ascertained through two distinct experimental setups: interfacing with an Arduino Uno as a slave device and the deployment of an air quality monitoring system leveraging the SPI peripheral.    **4.3.1 SPI Module and Arduino Uno Interaction**  The configuration involving the Arty A7 board as the master and the Arduino Uno as the slave is presented in Figure 4.3. This setup required connecting the SPI module pins to the corresponding pins on the Arduino, following which, programming was carried out to enable data reception on the Arduino, substantiated by the 32-bit data display on the serial monitor as shown in Figure 4.4. This scenario exemplifies the bidirectional capability and reliability of the SPI module in real-world applications.    **4.3.2 Application with an Air Quality Sensor**  An innovative test involved the SPI module’s interface with the MQ135 air quality sensor, depicted in Figure 4.5. The sensor integration aims to monitor air quality, with an operational mechanism set to trigger an LED based on predefined analog output thresholds. The successful interaction confirms not only the SPI module’s versatile application but also highlights its potential contribution to environmental monitoring initiatives. The sensor’s output, visible on the serial monitor (Figure 4.6), reinforces the accuracy and real-time response capability of the SPI-enabled monitoring system. |

**Chapter 5**

**Conclusions and future scope**

his chapter provides a synthesis of the project's achievements and outlines the path forward, highlighting the project’s closure, key accomplishments, and the envisioned future developments in the domain of Serial Peripheral Interface (SPI) within System on Chip (SoC) architectures.

**5.1 Conclusion**

This project embarked on designing and implementing an interface between the processor and peripheral devices, with a keen emphasis on the Serial Peripheral Interface (SPI). The venture was twofold: to dissect the operational dynamics of SPI and to delve into the design intricacies associated with its implementation. By employing Verilog HDL for the SPI's implementation and the Universal Verification Methodology (UVM) for its verification, we mapped out the foundational elements of SPI—its operational principles, the configuration of its registers, and the initiation and management of serial communication between the master and selected slave devices. The project successfully delineated a comprehensive process for setting up an efficient SPI communication environment, alongside providing a detailed overview of the UVM-based verification platform for rigorously testing the SPI Design Under Test (DUT).

**5.2 Future scope**

Looking ahead, the project opens several avenues for broader application and deeper exploration within the field of RISC-V based SoC Design. The adaptability and efficiency of the SPI protocol evident from this project underscore its potential for broader applications, ranging from simple device control to complex data transfer systems within diverse SoC configurations. The project’s future directions include.

Chapter 5

Conclusions and future scope

This is the concluding chapter that includes project closure and epilogue along with the

future scope of our project.

5.1 Conclusion

The processor to device interface is created and used in this project. The performance

analysis of the Serial Peripheral Interface is the primary focus, along with the design effort.

We demonstrated how a Serial Peripheral Interface was implemented and verified. It has been

demonstrated how to build and verify a configurable module of the SPI protocol using UVM

for verification and Verilog HDL for implementation. SPI’s fundamental features and operation

are covered, along with an explanation of registers, signals, and pins. It explains how to set

up a serial communication environment between the chosen slave device and the master. The

description of the UVM verification platform for the SPI design under test (DUT) is included

in the functional verification of SPI.

5.2 Future scope

The future scope is to test on various applications and back-end flow for RISC-V based

SoC Design.