ROCKSAVAGE
TECHNOLOGY

Spi
Product User Guide

rocksavagetech.chiselWare.Spi

IPF certified to level: **0** of 5

Abdelrahman Abbas, Ahmed Elmenshawi, Nick Allison, Jimmy Bright

March 27, 2025

Contents

1 Errata and Known Issues

1.1 Errata

None.

1.2 Known Issues

None.

2 Port Descriptions

2.1 Spi Interface

The ports for **Spi** are shown below in Table 1. The width of several ports is controlled by the following input parameters:

Port Name	Width	Direction	Description
sclk	1	Input/Output	Generated as Output by the Master to drive transactions to Slave. Taken as Input by Slave to respond to transactions
miso	1	Input/Output	Input data for Master config. and Output data for Slave config.
mosi	1	Input/Output	Output data for Master config. and Input data for Slave config.
cs	1	Input/Output	Driven as Output by Master to select Slave. Taken as Input by Slave to be activated

Table 2: Spi Ports Descriptions

2.2 Apb3 Interface

The **Apb3 Interface** is a regular Apb3 Slave Interface. All signals supported are shown below in Table 2. See the *AMBA Apb Protocol Specifications* for a complete description of the signals. The width of several ports is controlled by the following input parameters:

- *dataWidth* is the width of PWDATA and PRDATA in bits
- *addrWidth* is the width of PADDR in bits

Port Name	Width	Direction	Description
PCLK	1	Input	Positive edge clock
PRESETN	1	Input	Active low reset
PSEL	1	Input	Indicates slave is selected and a data transfer is required
PENABLE	1	Input	Indicates second cycle of Apb transfer
PWRITE	1	Input	Indicates write access when HIGH and read access when LOW
PADDR	*addrWidth*	Input	Address bus
PWDATA	*dataWidth*	Input	Write data bus driven when PWRITE is HIGH
PRDATA	*dataWidth*	Output	Read data bus driven when PWRITE is LOW
PREADY	1	Output	Transfer ready
PSLVERR	1	Output	Transfer error

Table 4: Apb Ports Descriptions

3 Parameter Descriptions

The parameters for **Spi** are shown below in Table 3.

Name	Type	Min	Max	Description
dataWidth	Int	1	≤ 32	The data width of PWDATA, and PRDATA. Can be 8, 6, or 32 bits wide
addrWidth	Int	1	≤ 32	The Apb address bus width

Table 6: Parameter Descriptions

The Spi is instantiated into a design as follows:

```
// Valid Spi Instantiation Example
val mySpi = new Spi(
  dataWidth = 32,
  addrWidth = 32 )
```

4 Operating Modes

The SPI core's versatility is exemplified by its ability to operate in both master and slave modes. Each mode offers distinct functionalities and configurations to cater to diverse system requirements.

4.1 Master Mode

In master mode, the SPI core assumes control over the communication process. It is responsible for generating the serial clock (SCK) and managing the Slave Select (SS) lines to initiate and terminate communication with slave devices.

4.1.1 Normal Mode

Operating in normal mode without buffering involves the following characteristics:

- **Immediate Transmission:** Data transmission begins as soon as data is written to the Data Register (`DATA`).
- **Single-Buffered Transmission:** The SPI core uses a single buffer for transmission, meaning only one byte can be sent at a time.
- **Double-Buffered Reception:** Reception is handled using a double buffer, allowing for more efficient data handling without data loss.
- **Write Collision Detection:** If an attempt is made to write to `DATA` before the current transmission completes, the Write Collision flag (`WRCOL`) is set, indicating a collision has occurred.

4.1.2 Buffer Mode

Enabling buffer mode introduces additional buffering capabilities, enhancing data handling efficiency:

- **Double-Buffered Transmission:** Allows two bytes to be buffered for transmission, enabling continuous data flow without waiting for the previous transmission to complete.
- **Triple-Buffered Reception:** Reception is managed using three buffers, providing ample space to handle incoming data without overflow.
- **Interrupt-Driven Operations:** Buffer mode facilitates the use of interrupts for efficient data handling. The Data Register Empty Interrupt Flag (`DREIF`) can be used to determine when `DATA` is ready to accept new data.
- **Enhanced Throughput:** By allowing multiple bytes to be buffered, buffer mode significantly increases the data throughput, making it suitable for high-speed data transfers.

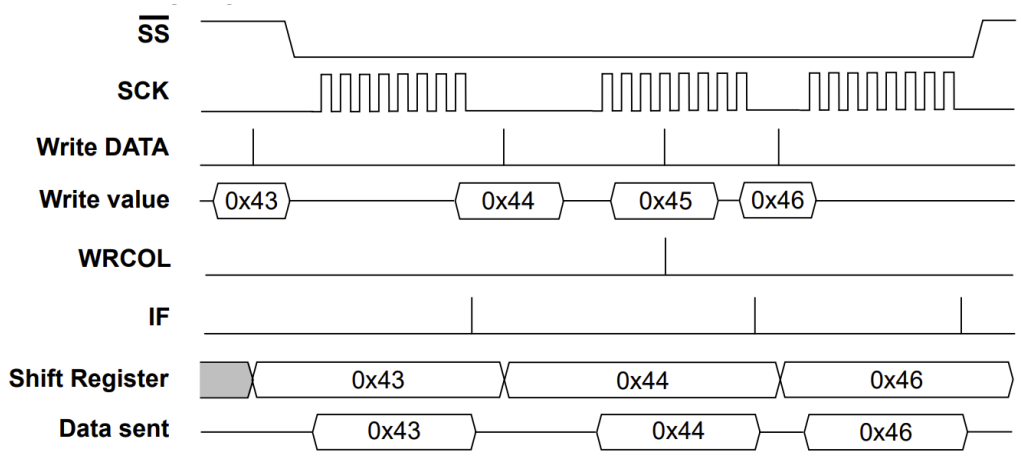

Figure 1: Master Mode Timing Diagram

4.1.3 Normal Mode

In normal mode without buffering:

- **Transmission starts immediately after writing to `DATA`.**
- **The core is single-buffered for transmission and double-buffered for reception.**
- **The Write Collision flag (`WRCOL`) is set if `DATA` is written before a transmission completes.**

4.1.4 Buffer Mode

Enabling buffer mode provides additional buffering:

- **Double-buffered transmission and triple-buffered reception.**
- **Allows writing to `DATA` while a transmission is ongoing.**
- **Use the Data Register Empty Interrupt Flag (`DREIF`) to check if `DATA` can be written.**

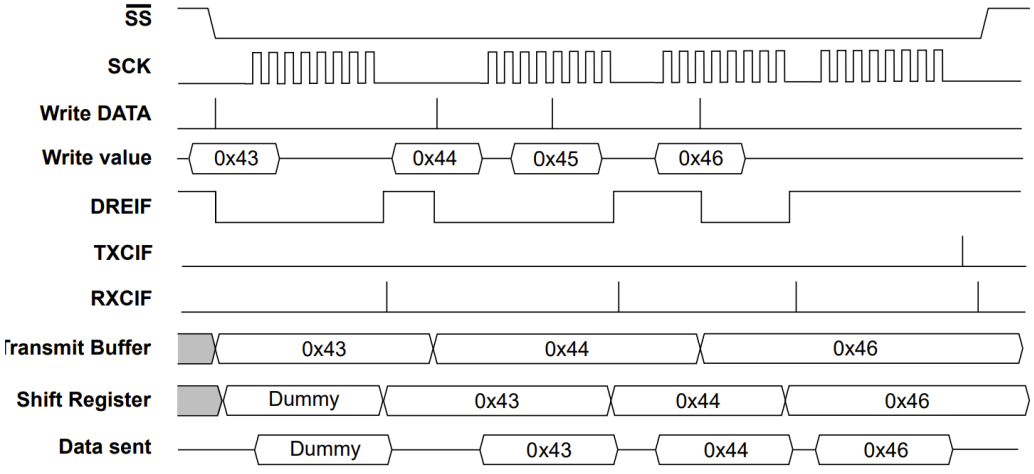

Figure 2: SPI Buffer Mode Operation

4.2 Slave Mode

In slave mode, the SPI core operates passively, awaiting instructions from the master device. It does not generate the clock signal but instead responds to the master's SCK and SS signals.

4.2.1 Normal Mode

Operating in normal mode without buffering involves the following characteristics:

- **Initiated by Master:** Transmission and reception are controlled by the master device, which dictates when data transfers occur.
- **Data Preparation:** Data must be written to the Data Register (`DATA`) before the master initiates a transfer. Failure to do so can result in incomplete or corrupted data transmission.
- **Write Collision Detection:** Similar to master mode, if `DATA` is written during an ongoing transfer, the Write Collision flag (`WRCOL`) is set, indicating a collision.

4.2.2 Buffer Mode

Buffer mode in slave operation offers enhanced data handling capabilities:

- **Pre-Buffered Transmission:** Allows the slave to prepare multiple bytes of data in advance, ensuring seamless data transmission when the master initiates a transfer.
- **Graceful Overrun Handling:** Buffer mode can detect and manage data overruns, preventing data loss and maintaining system stability.
- **Configurable Buffer Behavior:** The Buffer Mode Wait for Receive (`BUFWR`) bit in `CTRLB` allows configuration of how the buffer behaves during data reception, enabling flexible data handling strategies.

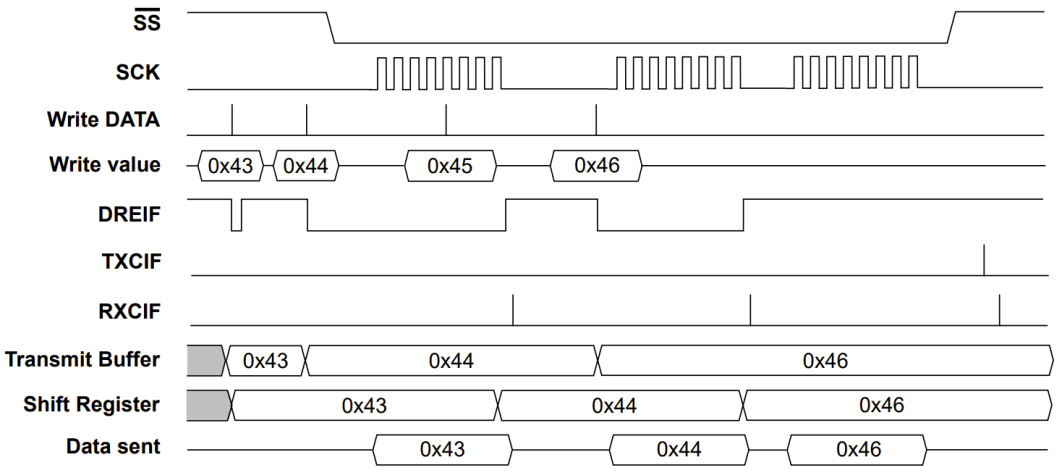

Figure 3: Slave Mode Timing Diagram

4.2.3 Normal Mode

In normal mode:

- **Transmission begins when SS is low and SCK pulses are received.**
- **Data must be written to `DATA` before the master initiates a transfer.**
- **The Write Collision flag (`WRCOL`) is set if `DATA` is written during an ongoing transfer.**

4.2.4 Buffer Mode

Buffer mode in slave operation allows for:

- **Preparing data in advance for transmission.**
- **Handling data overruns more gracefully.**
- **Configurable behavior using the Buffer Mode Wait for Receive (`BUFWR`) bit.**

4.3 Data Transfer Modes

The SPI protocol defines four distinct data transfer modes, each determined by the Clock Polarity (CPOL) and Clock Phase (CPHA). These modes dictate the timing relationship between the clock signal and data signals, ensuring synchronized and accurate data transmission.

Table 7: SPI Data Transfer Modes

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

- **Mode 0 (CPOL=0, CPHA=0):**
  - **Clock Polarity (CPOL):** The clock signal (SCK) is low when idle.
  - **Clock Phase (CPHA):** Data is sampled on the leading (rising) edge of the clock.
  - **Description:** Data is set up on the trailing edge and sampled on the leading edge, ensuring data stability before sampling.
- **Mode 1 (CPOL=0, CPHA=1):**
  - **Clock Polarity (CPOL):** The clock signal (SCK) is low when idle.
  - **Clock Phase (CPHA):** Data is sampled on the trailing (falling) edge of the clock.
  - **Description:** Data is set up on the leading edge and sampled on the trailing edge, providing flexibility in data timing.
- **Mode 2 (CPOL=1, CPHA=0):**
  - **Clock Polarity (CPOL):** The clock signal (SCK) is high when idle.
  - **Clock Phase (CPHA):** Data is sampled on the leading (falling) edge of the clock.
  - **Description:** Data is set up on the trailing edge and sampled on the leading edge, suitable for systems where a high idle clock state is preferred.
- **Mode 3 (CPOL=1, CPHA=1):**
  - **Clock Polarity (CPOL):** The clock signal (SCK) is high when idle.
  - **Clock Phase (CPHA):** Data is sampled on the trailing (rising) edge of the clock.
  - **Description:** Data is set up on the leading edge and sampled on the trailing edge, allowing for alternate data sampling strategies.

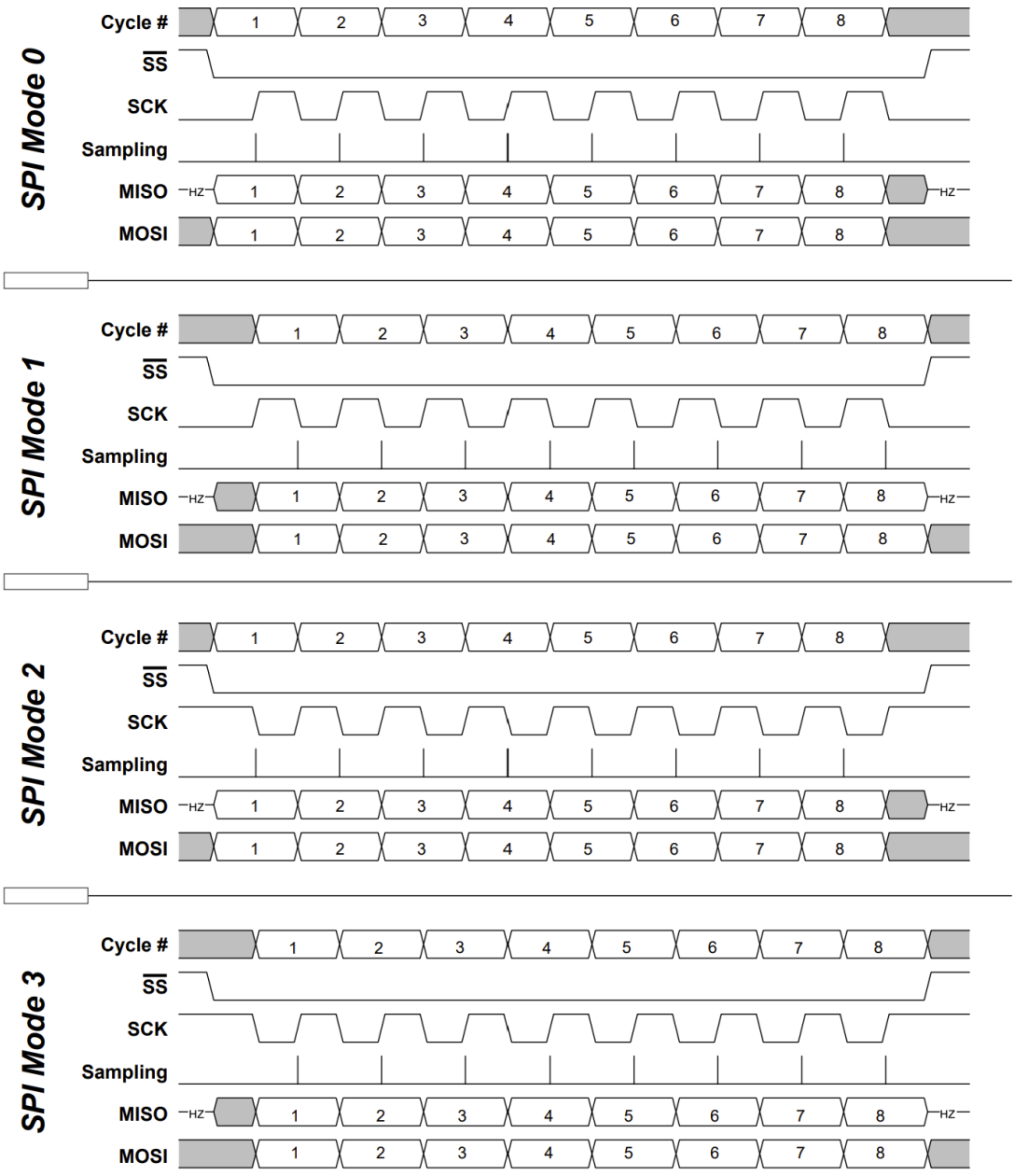

Figure 4: SPI Data Transfer Modes Timing Diagrams

5 Theory of Operations

The SPI (Serial Peripheral Interface) core is equipped with a variety of features that make it a versatile and efficient choice for synchronous serial communication in embedded systems. These features are described in-depth in the Register Interface section, but here is a brief overview:

- **Full Duplex, Three-Wire Synchronous Data Transfer:** Enables simultaneous transmission and reception of data, enhancing communication efficiency.
- **Master or Slave Operation:** The SPI core can function as either a master, initiating and controlling communication, or as a slave, responding to the master's commands.
- **LSb First or MSb First Data Transfer:** Flexibly configures the data transmission order based on the system requirements, supporting both Least Significant Bit (LSb) first and Most Significant Bit (MSb) first formats.
- **Programmable Bit Rates:** Allows customization of the SPI clock speed to match the performance needs of different peripherals and system constraints.
- **End of Transmission Interrupt Flag:** Notifies the system when a data transmission cycle is complete, facilitating efficient interrupt-driven communication.
- **Write Collision Protection:** Prevents data corruption by detecting and handling scenarios where multiple write attempts occur simultaneously.
- **Wake-up from Idle Mode:** Supports low-power operations by enabling the SPI core to wake up from idle states upon specific events or triggers.
- **Double-Speed Master SPI Mode:** Increases data throughput by doubling the SPI clock rate in master mode, enhancing overall system performance.

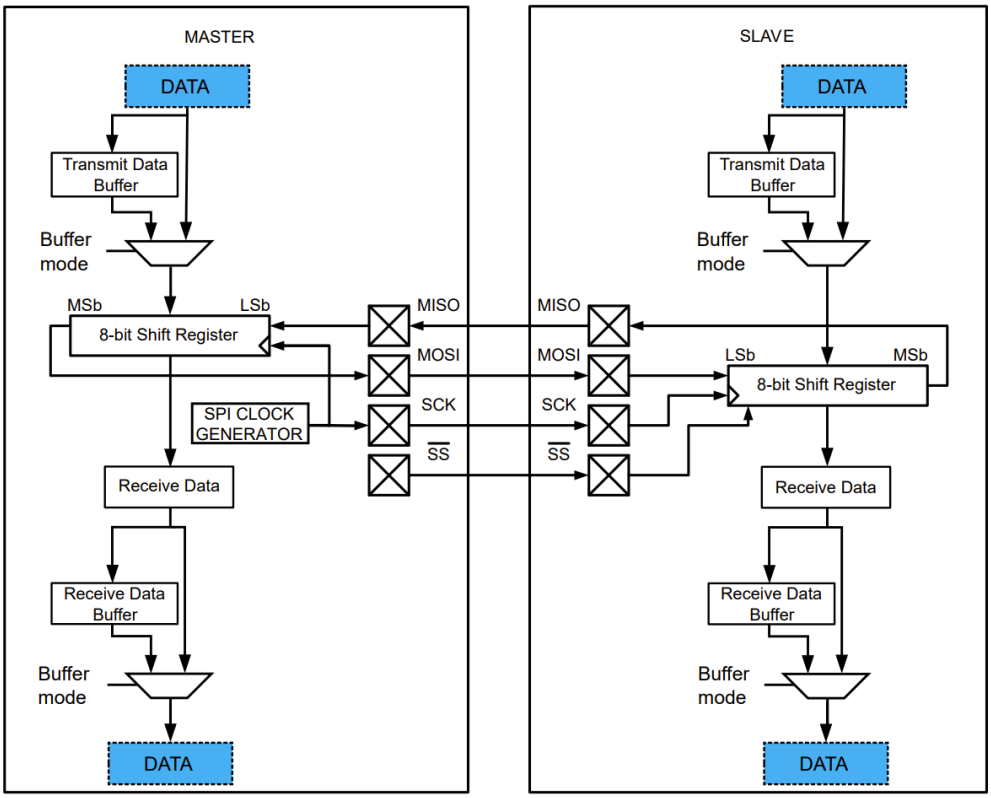

Figure 5: SPI Core Block Diagram

5.1 Interface Timing

Spi has a synchronous Apb3 interface, and a Spi interface. The timing diagram shown below in Figure 6 represents an instantiation with the following parameters.

```
val mySpi = new Spi(
  dataWidth = 8,
  addrWidth = 8 )
```

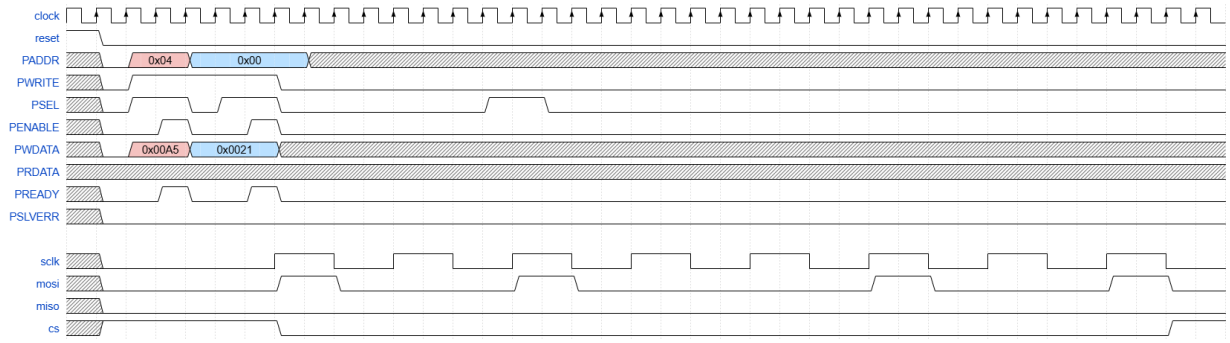

Figure 6: Timing Diagram

This shows the operation of enabling the Spi as a Master following the Apb protocol. Register CTRLA is written to configure the Master as MSB first, prescaler set at 4, and no double clock speed. Since CTRLB is not written to, and its default values are 0, there will be no buffer mode, and CPOL/CPHA mode set to 0. The DATA register is written with 0xA5, which is the data the Master will transmit to the Slave.

The *sclk* port is driven once the master is enabled via CTRLA. Since CPOL/CPHA is set to 0, *sclk* begins low, and sampling happens on the rising edge. *mosi* shifts out the value written to DATA, one bit at a time, MSB first. *miso* stays low since the Slave is not sending any data to the Master. *cs* is driven low once transmission begins, signalling to the Slave that a transaction is occuring.

Once the transaction is complete, all the SPI ports go low.

5.2 Register Interface

When programming registers, each register starts on a byte address, and the last bits it would take up in its final byte based on its size are unused. To find the size in bytes for any register, divide by the register size, and round up to the nearest whole number. For example, a 32-bit register would take up 4 bytes, and a 1-bit register would take up 1 byte.

Name	Size (Bits)	Description
CTRLA	8	Clock and Transmission Control Features
CTRLB	8	Transmission Mode Control
INTCTRL	8	Enable Flags for Interrupts
INTFLAGS	8	Interrupt Status Register
DATA	dataWidth	Not a Physical Register. Connects to the shift register for transmitting and recieving.

Table 9: Register Interface

5.3 Register Description

5.4 Control Register A (CTRLA)

Table 10: Control Register A (CTRLA)

Bit	7	6	5	4	3	2-1	0
Name	N/A	DORD	MASTER	CLK2X	N/A	PRESC[1:0]	ENABLE

- **Bit 6 - DORD (Data Order):**
  - 0: MSB (Most Significant Bit) transmitted first.
  - 1: LSb (Least Significant Bit) transmitted first.

  Description: This bit configures the order in which data bits are transmitted and received. Selecting LSb first can be beneficial for certain peripherals or communication protocols that prioritize lower-order bits.
- **Bit 5 - MASTER:**
  - 0: Slave mode.
  - 1: Master mode.

  Description: Determines whether the SPI core operates as a master or a slave. In master mode, the core generates the clock and controls SS lines. In slave mode, it responds to the master's commands.
- **Bit 4 - CLK2X (Clock Double):**
  - 0: SPI clock rate not doubled.
  - 1: SPI clock rate doubled.

  Description: When set, this bit doubles the SPI clock frequency after prescaling in master mode, effectively increasing data transfer speed.
- **Bits 2-1 - PRESC[1:0] (Prescaler):**
  - 00: Divide by 4.
  - 01: Divide by 16.
  - 10: Divide by 64.
  - 11: Divide by 128.

  Description: These bits define the base division factor applied to the peripheral clock to generate the SPI clock rate. Higher division factors result in lower SPI clock speeds.
- **Bit 0 - ENABLE:**
  - 0: SPI disabled.
  - 1: SPI enabled.

  Description: Activates or deactivates the SPI core. Disabling the SPI core can save power when communication is not required.

5.5 Control Register B (CTRLB)

Table 11: Control Register B (CTRLB)

Bit	7	6	5	4	3-2	1-0
Name	BUFEN	N/A	N/A	EIGHTO	EIGHTB[3:2]	MODE[1:0]

- **Bit 7 - BUFEN (Buffer Enable):**
  - 0: Buffer mode disabled.
  - 1: Buffer mode enabled.

  Description: When set, enables buffer mode, allowing the SPI core to handle multiple data bytes efficiently through additional buffering mechanisms.

- **Bit 4 - EIGHTO (8 Bit Only):**
  - `0`: Data is shifted as normal.
  - `1`: Data is forced to only shift upper 8-bits or lower 8-bits (depending on CTRLA DORD configuartion)

  Description: Useful for 16 or 32-bit cores to only shift 8 bits. Can only be used during Normal mode - disable this if you wish to use buffer mode.
- **Bits 3-2 - EIGHTB[3:2] (8 Bit Transactions):**
  - `00`: Data is not split into 8-bit transactions.
  - `01`: Used for 16-bit configuration to split into 8-bit transactions.
  - `10`: Used for 32-bit configuration to split into 8-bit transactions.
  - `11`: N/A.

  Description: Used to split 16-bit and 32-bit into 8-bit transactions. Note: Must be 00 for an 8-bit core. 16-bit core can ONLY use 01, 32-bit core can ONLY use 10.
- **Bits 1-0 - MODE[1:0] (Mode Selection):**
  - `00`: Mode 0.
  - `01`: Mode 1.
  - `10`: Mode 2.
  - `11`: Mode 3.

  Description: Selects the SPI data transfer mode based on CPOL and CPHA settings, defining the timing relationship between the clock and data signals.

5.6 Interrupt Control Register (`INTCTRL`)

Table 12: Interrupt Control Register (`INTCTRL`)

Bit	7	6	5	4	3-1	0
Name	N/A	TXCIE	N/A	N/A	N/A	IE

- **Bit 6 - TXCIE (Transfer Complete Interrupt Enable):**
  - `0`: Transfer Complete interrupt disabled.
  - `1`: Transfer Complete interrupt enabled.

  Description: Enables the generation of an interrupt when a data transmission cycle is complete in buffer mode, signaling that the transmit buffer is empty and ready for new data.
- **Bit 0 - IE (Interrupt Enable):**
  - `0`: General interrupts in normal mode disabled.
  - `1`: General interrupts in normal mode enabled.

  Description: Controls the overall interrupt functionality in normal mode. When set, the SPI core can generate interrupts based on the `IF` flag in normal mode.

5.7 Interrupt Flags Register (`INTFLAGS`)

The Interrupt Flags Register (`INTFLAGS`) monitors various interrupt conditions and flags. Its behavior varies depending on whether the SPI core is operating in normal mode or buffer mode.

5.7.1 Normal Mode

In normal mode, the `INTFLAGS` register contains the following flags:

Table 13: Interrupt Flags Register (`INTFLAGS`) - Normal Mode

Bit	**Name**
7	IF
6	WRCOL

- **Bit 7 - IF (Interrupt Flag):**
  - `0`: No interrupt.
  - `1`: Interrupt flag set, indicating a data transfer has completed.

  Description: This flag is set automatically when a data transfer cycle is finished. It can be cleared by executing the corresponding interrupt vector or by reading the `INTFLAGS` register followed by accessing the `DATA` register.
- **Bit 6 - WRCOL (Write Collision Flag):**
  - `0`: No write collision detected.
  - `1`: Write collision occurred, indicating an attempt to write to `DATA` during an ongoing transfer.

  Description: This flag alerts the system to potential data corruption caused by simultaneous write attempts. It must be cleared by first reading the `INTFLAGS` register and then accessing the `DATA` register.

5.7.2 Buffer Mode

In buffer mode, the `INTFLAGS` register encompasses additional flags to manage enhanced data handling:

Table 14: Interrupt Flags Register (`INTFLAGS`) - Buffer Mode

Bit	7	6	5	4	0
Name	N/A	TXCIF	DREIF	N/A	BUFOVF

- **Bit 6 - TXCIF (Transfer Complete Interrupt Flag):**
  - `0`: Transfer not complete.
  - `1`: Transfer complete and transmit buffer is empty.

  Description: Signals that all data in the transmit shift register has been shifted out, and there is no new data pending in the transmit buffer. This flag is cleared by writing a `1` to its bit position.
- **Bit 5 - DREIF (Data Register Empty Interrupt Flag):**
  - `0`: `DATA` register is ready to accept new data.
  - `1`: `DATA` register not ready for new data.

  Description: Indicates that the `DATA` register is empty and ready to receive new data for transmission. Writing new data to `DATA` clears this flag, either by writing the data or disabling the interrupt.
- **Bit 0 - BUFOVF (Buffer Overflow Flag):**
  - `0`: No buffer overflow.
  - `1`: Buffer overflow detected.

  Description: Indicates that a buffer overflow has occurred during data reception, meaning the receive buffer was full when a new byte arrived. This flag is cleared by reading the `DATA` register or writing a `1` to its bit position.

5.8 Data Register (`DATA`)

The Data Register (`DATA`) serves as the primary interface for sending and receiving data through the SPI core.

- **Writing to `DATA`:**
  - In **master mode**, writing a byte to `DATA` initiates the transmission process. The byte is shifted out via the MOSI line while simultaneously receiving data from the slave through the MISO line.
  - In **slave mode**, writing to `DATA` prepares the data to be sent to the master during the next communication initiated by the master.
- **Reading from `DATA`:**
  - Reading from `DATA` retrieves the most recently received byte of data.
- Note: DATA is not a physical register. Depending on what mode is configured, it is mapped to other registers as described below.
- **Behavior in Different Modes:**
  - **Normal Mode:**
    - `DATA` writes go directly to the shift register for immediate transmission.

    - `DATA` reads fetch data from the Receive Data register.
- **Buffer Mode:**
    - `DATA` writes are stored in the Transmit Data Buffer register, allowing for buffered transmission.
    - `DATA` reads retrieve data from the Receive Data Buffer register, ensuring that multiple received bytes can be handled efficiently.

5.9 Register Addresses

Register Name	Address Start	Address End
CTRLA	0x0	0x0
CTRLB	0x1	0x1
INTCTRL	0x2	0x2
INTFLAGS	0x3	0x3
DATA	0x4	0x4

Table 15: 8-bit Register Addressing

dataWidth: 8

Register Name	Address Start	Address End
CTRLA	0x0	0x0
CTRLB	0x1	0x1
INTCTRL	0x2	0x2
INTFLAGS	0x3	0x3
DATA	0x4	0x5

Table 16: 16-bit Register Addressing

dataWidth: 16

Register Name	Address Start	Address End
CTRLA	0x0	0x0
CTRLB	0x1	0x1
INTCTRL	0x2	0x2
INTFLAGS	0x3	0x3
DATA	0x4	0x6

Table 17: 32-bit Register Addressing

dataWidth: 32

6 Simulation

6.1 Tests

The test bench generates a number (default is 2) configurations of the SPI that are highly randomized. The user can run the full test-suite with n-configurations using "sbt "test" -DtestName="regression"".
User can also run individual tests or test groups by substituting "regression" with the relevant test name. A description of the tests is below:

Test Group Name	Test Name	Test Type	Test Description
transmitTests	masterMode	Directed	Configures SPI as Master and Tests Transmit w/ Random Data
transmitTests	slaveMode	Directed	Configures SPI as Slave and Tests Transmit w/ Random Data
transmitTests	fullDuplex	Random	Configures one SPI as Master, other as Slave. Tests transmission w/ Random Data Across all 4 CPOL/CPHA Modes
transmitTests	bitOrder	Directed	Should Transmit and Recieve Random Data Correctly in MSB and LSB modes
transmitTests	dataOrder	Directed	Should partition 16-bit data into two 8-bit transactions
transmitTests	dataOrderBuffer	Random	Should partition 16-bit data into two 8-bit transactions with buffer enabled
clockTests	prescaler	Directed	Clock Speed Test for All Prescaler Values
clockTests	doubleSpeed	Directed	Clock Speed Test with Double Speed Enabled
interruptTests	txComplete	Directed	Test Transmission Complete Interrupt Flag
interruptTests	wcolFlag	Directed	Check if Write Collision Flag is Triggered in Normal Mode
interruptTests	dataEmpty	Directed	Check Data Register Empty Flag Triggers in Buffer Mode
interruptTests	overFlow	Random	Check Buffer Mode Operation still Continue with Overflow, and Overflow Flag is Set
modeTests	bufferTx	Directed	Verify Data is Transmitted and Recieved Correctly in Buffer Mode
modeTests	normalRx	Directed	Check Recieve Register Operation in Normal Mode
modeTests	daisyChain	Random	Check SPI Operation w/ 1 Master and 2 Slaves in Daisy Chain in Normal Mode
modeTests	daisyChainBuffer	Random	Check SPI Operation w/ 1 Master and 2 Slaves in Daisy Chain in Buffer Mode

Table 18: Test Suite

6.2 Toggle Coverage

Current Score: 87.32%
All inputs and outputs are checked to insure each toggle at least once. If coverage is enabled during core instantiation, a cumulative coverage report of all tests is generated under ./out/cov/verliog.
Exceptions are higher bits of the *PADDR*, *PWDATA*, and *PRDATA* which are intended to be static during each simulation. These signals are excluded from coverage checks.

6.3 Code Coverage

Current Score: 94.37%
Code coverage for all tests can be generated as follows:

```
$ sbt coverage test -DtestName="allTests"
$ sbt coverageReport
$ python3 -m http.server 8000 --directory target/scoverage-report/
View report on local host: http://localhost:8000/index.html
```

6.4 Running simulation

Simulations can be run directly from the command prompt as follows:

```
$ sbt test -DtestName="allTests"
```

or from make as follows:

```
$ make test
```

6.5 Viewing the waveforms

The simulation generates an FST file that can be viewed using a waveform viewer. The command to view the waveform is as follows:

```
$ gtkwave ./out/test/Spi.fst
```

7 Synthesis

7.1 Area

The Spi has been tested in a number of configurations and the following results should be representative of what a user should see in their own technology.

Config Name	gpioWidth	numVirtualPorts	sizeOfVirtualPorts	Gates
8_8_8	8	8	8	6734
16_8_8	16	8	8	9325
32_8_8	32	8	8	14981

Table 20: Synthesis results

7.2 SDC File

An `.sdc` file is generated to provide synthesis and static timing analysis tools guidance for synthesis.
The `Spi.sdc` file is emitted and found in the `./out/sta` directory.

7.3 Timing

The following timing was extracted using the generated .sdc files using the Nangate 45nm free library.

Config Name	Period	Duty Cycle	Input Delay	Output Delay	Slack
8_8_8	5ns	50%	20%	20%	1.79 (MET)
8_8_8	5ns	50%	20%	20%	1.79 (MET)
8_8_8	5ns	50%	20%	20%	1.81 (MET)

Table 22: Static Timing Analysis results

7.4 Multicycle Paths

None.

8 Usage Examples

8.1 Full-Duplex Communication

The following example demonstrates full-duplex SPI communication in mode 1 (CPOL=0, CPHA=1):

```
// 1. Configure Master and Slave for mode 1
write(MASTER_CTRLB, 0x01);    // Set SPI mode 1 (CPOL=0, CPHA=1)
write(SLAVE_CTRLB, 0x01);     // Set same mode on slave

// 2. Prepare data for transmission
write(MASTER_DATA, 0xA5);     // Master will send 0xA5
write(SLAVE_DATA, 0x3C);      // Slave will send 0x3C

// 3. Enable Master and Slave
write(MASTER_CTRLA, 0x21);    // Enable Master, Enable SPI
write(SLAVE_CTRLA, 0x01);     // Enable Slave

// 4. Wait for transmission to complete
while (!(read(MASTER_INTFLAGS) & 0x80)); // Wait for RXCIF flag

// 5. Read received data
uint8_t master_rx = read(MASTER_DATA);  // Should be 0x3C
uint8_t slave_rx = read(SLAVE_DATA);    // Should be 0xA5
```

8.2 Changing Bit Order

The following example demonstrates LSB-first transmission:

```
// 1. Configure Master for LSB-first transmission
write(MASTER_CTRLB, 0x80);    // Set DORD bit (LSB first)

// 2. Prepare data
write(MASTER_DATA, 0x81);      // Binary: 10000001

// 3. Enable Master
write(MASTER_CTRLA, 0x21);     // Enable Master, Enable SPI

// 4. Expected transmission order:
//    Bit 0 (1) -> Bit 1 (0) -> ... -> Bit 7 (1)
```

8.3 Buffer Mode Operation

The following example demonstrates buffered transmission:

```
// 1. Enable buffer mode (must be done before writing data)
write(MASTER_CTRLB, 0x80);    // Enable buffer mode

// 2. Prepare multiple data bytes
write(MASTER_DATA, 0x12);     // First byte
write(MASTER_DATA, 0x34);     // Second byte (buffered)
write(MASTER_DATA, 0x56);     // Third byte (buffered)

// 3. Enable Master with interrupt
write(MASTER_INTCTRL, 0x40);  // Enable TX complete interrupt
write(MASTER_CTRLA, 0x21);    // Enable Master, Enable SPI
```

```
// 4. Check buffer status
while (!(read(MASTER_INTFLAGS) & 0x40)); // Wait for TXCIF flag

// 5. Transmission continues automatically with buffered data
```

8.4 Normal Reception with Interrupts

The following example demonstrates reception with interrupt:

```
// 1. Configure reception interrupt
write(MASTER_INTCTRL, 0x01);  // Enable RX complete interrupt

// 2. Enable Master
write(MASTER_CTRLA, 0x21);    // Enable Master, Enable SPI

// 3. In interrupt service routine:
void SPI_ISR() {
    if (read(MASTER_INTFLAGS) & 0x80) {  // Check RXCIF
        uint8_t data = read(MASTER_DATA);
        // Process received data...
        write(MASTER_INTFLAGS, 0x80);    // Clear flag
    }
}
```

8.5 Daisy Chain Configuration

The following example demonstrates daisy-chained SPI devices:

```
// 1. Configure all devices (master + 2 slaves)
write(MASTER_CTRLA, 0x21);    // Enable Master
write(SLAVE1_CTRLA, 0x01);    // Enable Slave 1
write(SLAVE2_CTRLA, 0x01);    // Enable Slave 2

// 2. Send data through the chain
write(MASTER_DATA, 0xAA);     // First byte for slave 1
write(MASTER_DATA, 0xBB);     // Second byte for slave 2

// 3. Wait for complete transmission
delay(SPI_TRANSFER_TIME * 3); // Wait for 3 full transfers

// 4. Read received data
uint8_t slave1_rx = read(SLAVE1_DATA);  // Should be 0xAA
uint8_t slave2_rx = read(SLAVE2_DATA);  // Should be 0xBB
uint8_t master_rx = read(MASTER_DATA);  // Returns data from slave 2
```