VERIFICATION OF SERIAL PERIPHERAL INTERFACE (SPI)

A Project Report
submitted by

PAWAN KUMAR
(EE18M053)

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.
JUNE 2020

CERTIFICATE

This is to certify that the project titled " **Verification of Serial Peripheral Interface (SPI) using Cocotb Verilator** " being submitted to the Indian Institute of Technology Madras by **Pawan Kumar (EE18M053)**, in partial fulfilment of the requirements for the award of the degrees of Master of Technology in Microelectronics and Photonics in Electrical Engineering is a bonafide record of work carried out by him/her under my supervision. The contents of this project report, in full or in parts, have not been submitted to any other institute or university for the award of any degree or diploma.

Guide	**Co-guide**
Prof- Kamakoti V.	**Prof.- Anbarasu M.**
Professor	Associate Professor
Compter Science Engg. Dept.	Electrical Engineering Dept.
IIT Madras, 600036	IIT Madras, 600036
Place – Chennai	Place - Chennai

AKNOWLEDGEMENT

I express my sincere thanks to Dr. Kamakoti V. and Dr. Anbarasu M. for their guidance and constant encouragement throughout the project work. I am grateful to them for providing me their valuable time through various sessions to discuss the project work which enabled me to take this project to fruitful completion.

Special thanks to Professor Kamakoti V. for giving me opportunity to work in Shakti Processo project, helped in understanding Digital Design and Implementation. I am greatly indebte- d to him for the knowledge on VLSI design that I gained during the course of my project work.

My sincere gratitude to Dr. Anbarasu .M for explaining the basics W/R of SRAM memories being act as slave for verifying Serial Periferal Interface.

I would also like to thank my Mentor - Lavanya Jagan for explaining me concepts of Verification plan and for helping me in learning the verification tools Cocotb-Verilator.

My sincere thanks to all my friends at IIT Madras, who supported me during my stay inthe campus and made it really enjoyable and memorable.

Finally, I thank my family for their support and constant encouragement.

Pawan Kumar
EE18M053

Content
Page Number

List of Figures

1. Introduction

Verification of SPI

The rapid development of modern integrated circuits not only increased the complexity of integrated circuit (IC) design, but also made the IC verification equally challenging. Around 70% to 80% of the entire design cycle time is allotted to verification, and traditional verification methodologies are no longer able to support current verification requirements . In 2002, the Accellera Systems Initiative released SystemVerilog (SV) a a unified hardware design and verification language. SystemVerilog language was an amalgamation of constructs from different languagessuch as Vera, Super Log, C, Verilog and VHDL languages. Moreover, in 2005 IEEE standard-ized (1800-2005) SystemVerilog. SystemVerilog supports behavioral, register transfer level, and gate level descriptions. SystemVerilog also supports testbench development by the inclusion ofobject-oriented constructs, cover groups, assertions, constrained random constructs, applicationspecific interface to other languages .Universal Verification Methodology (UVM) is a standardized verification methodology fortestbench creation an is derived form the Open Verification Methodology (OVM), and also in-herits some features from Verification Methodology Manual (VMM). Use of the UVM standardenables an increase in verification productivity by creating a reusable verification platform and verification components. In this Project I have used Cocotb-verilator an open source for bloack level verification of SOC i.e. SPI .The verification results of this work show the effectiveness and feasi-bility of the proposed verification environment System on Chip (SoC) is used for intelligent control feature with all the integrated compo-nents connected to each other in a single chip. To comlete a full system, every SoC must belinked to other system components in an efficient way that allows a faster error-free communi-cation. Data communication between core controller modu les and other external devices likeexternal SRAM, FLASH, EEPROMs, DACs, ADCs. is critical. Different forms of communication protocol exist such as high throughput protocols like Ethernet,USB, SATA PCI-Express which are usedfor data exchanges between whole systems. The Serial Peripheral Interface (SPI) is often con-sidered as light weight communication protocol. The primary purpose of the protocol is that itis suited for communication between integrated circuits for low and medium data transfer rateswith onboard peripherals and the serial bus provides a significant cost advantage.

The SPI interface can be used to communicate with external devices using the SPI protocol. The serial peripheral interface (SPI) protocol supports half-duplex, full duplex and simplex synchronous, serial communication with external devices. Theinterface can be configured as master, and in this case, it provides the communication clock (SCK) to the external slave device. The interface is also capable of operatingin a multi-master configuration.

1.0 Project Goal

The goal of this research work is to build a effective test bench that validates the SPI master controller with the help of the AXI4 Lite bus function model and SPI slave model. Cocotb Verilator is used for Verification of SPI The goal is achieved with the following objectives:

• To understand SPI protocol architecture and AXI4 Lite specific requirements, to establish a connection between the test bench components and core controller.

• To apply open source verification techniques such as Cocotb Verilator and Coverage Driven Functional Verification.

• To develop a reusable Verification IP for AXI4 Lite compliant SPI master core.

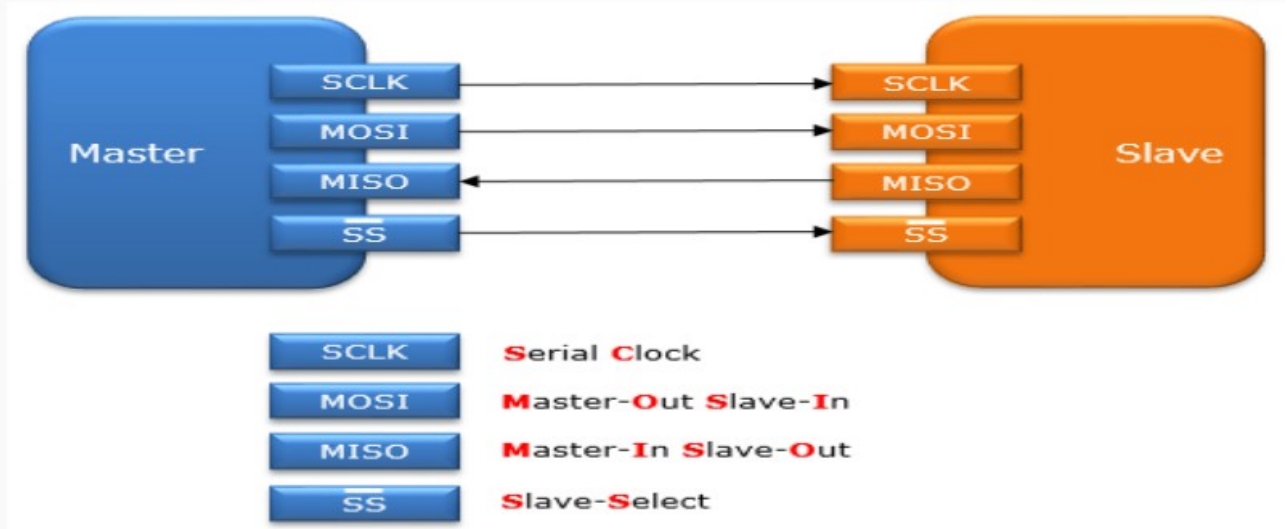


Fig.1 SPI Master Slave Interface

1.1 SPI main features

• Master or slave operation

• Full-duplex synchronous transfers on three lines

• Programmable clock polarity and phase

• Programmable data order with MSB-first or LSB-first shifting

• SPI bus busy status flag

1.2 SPI functional description

1.2.1 General Description

The SPI allows synchronous serial communication between the MCU and external devices. Application software can manage the communication by polling the status flag or using dedicated SPI interrupt.

Four I/O pins are dedicated to SPI communication with external devices.

• MISO: Master In / Slave Out data. In the general case, this pin is used to transmit data in slave mode and receive data in master mode.

• MOSI: Master Out / Slave In data. In the general case, this pin is used to transmit data in master mode and receive data in slave mode.

• SCK: Serial Clock output pin for SPI masters and input pin for SPI slaves.

• NSS: Slave select pin. Depending on the SPI and NSS settings, this pin can be used to either:
– select an individual slave device for communication
– synchronize the data frame or
– detect a conflict between multiple masters

1.2.2 Communications between one master and one slave

The SPI allows the MCU to communicate using different configurations, depending on the device targeted and the application requirements. These configurations use 2 or 3 wires (with software NSS management) or 3 or 4 wires (with hardware NSS management). Communication is always initiated by the master.

Full-duplex Communication

By default, the SPI is configured for full-duplex communication. In this configuration, the shift registers of the master and slave are linked using two unidirectional lines between the MOSI and the MISO pins. During SPI communication, data is shifted synchronously on the SCK clock edges provided by the master. The master transmits the data to be sent to the slave via the MOSI line and receives data from the slave via the MISO line. When the data frame transfer is complete (all the bitsare shifted), the information between the master and slave is exchanged.

1.2.3 Communication Formats

During SPI communication, receive and transmit operations are performed simultaneously. The serial clock (SCK) synchronizes the shifting and sampling of the information on the data lines. The communication format depends on the clock phase, the clock polarity, and the data frame format. To be able to communicate together, the master and slaves devices must follow the same communication format.

Clock Phase and Polarity Controls

The clock can be set to one of the four basic configuration defined in Motorola spi specification. Any one timing relationships may be chosen by software, using the CPOL and CPHA bits in the SPIx CR1 register.

• The CPOL bit sets the polarity of the clock signal during the idle state.
• The CPHA bit controls the edge on which the SCK pin captures the first data bit transacted

Data are latched on each occurrence of this clock transition type.

SPI Mode	CPOL	CPHASE	SCK Polarity in Idle State	Edge on which SCK pin capture the first data bit transacted
0	0	0	Logic Low	Rising Edge
1	0	1	Logic Low	Falling Edge
2	1	0	Logic High	Falling Edge
3	1	1	Logic High	Rising Edge

Data Frame Format

The SPI shift register can be set up to shift out MSB-first or LSB-first, depending on the value of the LSB FIRST bit. The data frame size is chosen by using the TOTAL BITS TX for transmission and TOTAL BITS RX for the reception. During communication, only bits within the data frame are clocked and transferred

1.2.4 Configuring the SPI in master mode

In the master configuration, the serial clock is generated on the SCK pin.

Procedure

• Select the BR[2:0] bits to define the serial clock baud rate (see SPI CR1 register).
• Select the CPOL and CPHA bits to define one of the four relationships between the data transfer and the serial clock.
• Load SPI DR register with data to be transmitted

• Select transmit only mode or transmit and immediate receive or receive only mode by selecting RX START or RX IMM START bit in SPI CR2 register.

• Configure the LSBFIRST bit in the SPI CR1 register to define the frame format.

• Configure the TOTAL BITS RX and TOTAL BITS TX in SPI CR1 register depend on number of bits to be transmit or receive respectively.

• The MSTR and SPE bits must be set to start the transaction.

In this configuration the MOSI pin is a data output and the MISO pin is a data input.

Transmit sequence

The data byte is parallel-loaded into the shift register (from the internal bus) during the first-bit transmission and then shifted out serially to the MOSI pin MSB first or LSB first depending on the LSB FIRST bit in the SPI CR1 register. The TXE flag is set on the transfer of data from the Tx Buffer in the shift register, and an interrupt is generated if the TXEIE bit in the SPI CR2 register is set.

Receive sequence

For the receiver, when data transfer is complete:

• The data in the shift register is transferred to the RX Buffer and the RXNE flag is set.
• An interrupt is generated if the RXNEIE bit is set in the SPI CR2 register.

At the last sampling clock edge, the RXNE bit is set, a copy of the data byte received in the shift register is moved to the Rx buffer. When the SPI DR register is read, the SPI peripheral returns this buffered value.

1.2.5 Data transmission and reception procedures

Rx and Tx buffers
In reception, data are received and then stored into an internal Rx buffer while In transmission, data are first stored into an internal Tx buffer before being transmitted. Read access of the SPI DR register returns the Rx buffered value, whereas writing access to the SPI DR stores the written data into the Tx buffer.

Start sequence in master mode

• In bidirectional mode, when only transmitting (BIDIMODE = 0 & RX IMM START = 0 & RX START = 0)
– The sequence begins when data are written into the SPI DR register(DR 5register).
– The data are then parallel loaded from the Tx buffer into the 8-bit shift
register during the first bit transmission and then shifted out serially to the MOSI pin.
– No data are received.

• In bidirectional mode, when transmitting and receiving but not at the same
time (BIDIMODE = 0 & RX IMM START = 1 & RX START = 0)
– The sequence begins when data are written into the SPI DR register(DR 5register).
– The data are then parallel loaded from the Tx buffer into the 8-bit shift
register during the first bit transmission and then shifted out serially to the MOSI pin.
– After transmission finishzed,he received data on the MISO pin is shifted in serially to the 8-bit shift register and then parallel loaded into the SPI DR register (Rx buffer)

1.2.6 Status Flags

Three status flags are provided for the application to completely monitor the state of the SPI bus.

Tx buffer empty flag (TXE)

The TXE flag is set when transmission TXFIFO has enough space to store data to send. TXE flag is linked to the TXFIFO level. The flag goes high and stays high until the TXFIFO level is lower or equal to 1/2 of the FIFO depth. An interrupt can be generated if the TXEIE bit in the SPI CR2 register is set. The bit is cleared automatically when the TXFIFO level becomes higher than 1/2.

Rx buffer not empty (RXNE)

The RXNE flag is set depending on the FRXTH bit value in the SPI CR2 register:
• If FRXTH is set, RXNE goes high and stays high until the RXFIFO level is greater or equal to 1/4 FIFO Depth.
• If FRXTH is cleared, RXNE goes high and stays high until the RXFIFO level is greater than or equal to 1/2 FIFO Depth.
An interrupt can be generated if the RXNEIE bit in the SPI CR2 register is set.The RXNE is cleared by hardware automatically when the above conditions are no longer true.

Busy flag (BSY)

The BSY flag is set and cleared by hardware (writing to this flag does not affect).When BSY is set, it indicates that a data transfer is in progress on the SPI (the SPI bus is busy).
The BSY flag can be used in specific modes to detect the end of a transfer so that the software can disable the SPI or its peripheral clock before entering a low-power mode, which does not provide a clock for the peripheral. This avoids corrupting the last transfer.

The BSY flag is cleared under any one of the following conditions.:
• When the SPI is correctly disabled

• When a fault is detected in Master mode (MODF bit set to 1)

• In Master mode, when it finishes a data transmission and no new data is readyto be sent or receive.

1.2.7 SPI error flags

An SPI interrupt is generated if one of the following error flags is set and interrupt is enabled by setting the ERRIE bit.

Overrun flag (OVR)
An overrun condition occurs when a master or slave receives data, and the RX FIFO has not enough space to store this collected data.When an overrun condition occurs, the newly received value does not overwrite the previous one in the RX FIFO. The recently received value is discarded, and all data transmitted subsequently is lost.
Mode fault (MODF)
Mode fault occurs when the master device has its internal NSS signal (NSS pin in NSS hardware mode, or SSI bit in NSS software mode) pulled low. This automati cally sets the MODF bit. Master mode fault affects the SPI interface in the following ways:

• The MODF bit is set and an SPI interrupt is generated if the ERRIE bit is set.
• The SPE bit is cleared. This blocks all output from the device and disables theSPI interface.
• The MSTR bit is cleared, thus forcing the device into slave mode.

As a security, hardware does not allow the SPE and MSTR bits to be set while the MODF bit is set.

CRC error (CRCERR)

This flag is used to verify the validity of the value received when the CRCEN bit in the SPI CR1 register is set. The CRCERR flag in the SPI SR register is set if the value received in the shift register does not match the receiver SPI RXCRCR value.
The flag is cleared by the software.
TI mode frame format error (FRE)
A TI mode frame format error is detected when an NSS pulse occurs during an ongoing communication when the SPI is operating in slave mode and configures to conform to the TI mode protocol. When this error occurs, the FRE flag is set in the SPI SR register.

1.3 SPI interrupts

During SPI communication an interrupts can be generated by the following events:

• Transmit TXFIFO ready to be loaded

• Data received in Receive RXFIFO

• Master mode fault

• Overrun error

• TI frame format error

• CRC protocol error

Interrupts can be enabled and disabled separately.

Section 2 - SPI Registers

2.1 SPI Register

The peripheral registers can be accessed by words (32-bit).

2.1.1 SPI control register 1

Address offset: 0x00
Reset value: 0x00000000

31-24	16-23	15	14	13	12	11	10
TOTAL BITS_RX	TOTAL BITS_TX	BIDI MODE	BIDI OE	CRC EN	CRC NEXT	CRCL	Rx ONLY
rw	rw	rw	rw	rw	rw	rw	rw

9	8	7	6	3-5	2	1	0
SSM	SSI	LSB FIRST	SPE	BR [2:0]	MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw

Table 1: SPI CR1

Bits [31:24]
TOTAL BITS RX Expected Total number of bits to be received. Receive state will go to idle after sending this many bits. One can receive maximum128 bits. (Reading status register, Device ID and data, etc).

Bits [23:16]
TOTAL BITS TX Total number of bits to transmitted. After transmitting this many bits it Transmit state will got to idle. One can send maximum 128 bits. It include the instruction, read/write address and write data.

Bit 15 BIDI MODE
Bidirectional data mode enable. This bit enables half-duplex communication using common single bidirectional data line.
0: 2-line unidirectional data mode selected
1: 1-line unidirectional data mode selected

Bit 14 BIDI OE
Output enable in bidirectional mode This bit combined with the BIDI-MODE bit selects the direction of transfer in bi-direction mode.
0: receive-only mode (Output Disabled)
1: transmit-only mode (Output Enabled)
Bit 13 CRC EN
Hardware CRC calculation Enable.

0: CRC calculation disable
1: CRC calculation enable

Bit 12 CRC NEXT

Transmit CRC Next.
0: Next Transmit value is from Tx buffer
1: Next Transmit value is from Rx buffer

Bit 11 CRCL

CRC Length bit is set and cleared by software to select the CRClength.

Bit 10 RX

ONLY Receive only mode enabled. This bit enables simplex communication using a single unidirectional line to receive data exclusively.Keep BIDIMODE bit clear when receiving the only mode is active.This bit is also useful in a multi slave system in which this particular slave is not accessed, the output from the accessed slave is not corrupted.

Bit 9 SSM

Software Slave Management.
When the SSM bit is set, the NSS
pin input is replaced with the value from the SSI bit.
0: Software slave management disabled
1: Software slave management enabled

Bit 8 SSI

Internal Slave Select.
This bit has an effect only when the SSM bit
is set. The value of this bit is forced onto the NSS pin and the I/O
value of the NSS pin is ignored

Bit 7 LSB FIRST

Frame Format
0: data is transmitted/received with the MSB first
1: data is transmitted/received with the LSB first
Note This bit should not be changed when communication is ongoing.

Bit 6 SPE

SPI Enables
0:Peripheral disabled
1:Peripheral Enabled

Bits [5:3] BR [2:0]

Baud Rate Control.
000: fCLK/2
001: fCLK/4
010: fCLK/6
011: fCLK/8
100: fCLK/10
101: fCLK/12

110: fCLK/14
111: fCLK/16
Note This bit should not be changed when communication is ongoing.
Bit 2 MSTR
Master selection
0: Slave Configuration
1: Master Configuration

Note This bit should not be changed when communication is ongoing.

Bit 1 CPOL
Clock polarity
0: CLK is 0 when idle
1: CLK is 1 when idle

Note This bit should not be changed when communication is ongoing.

Bit 0 CPHA
Clock Phase
0: The first clock transition is the first data capture edge
1: The second clock transition is the first data capture edge

Note This bit should not be changed when communication is ongoing.

2.1.2 SPI Control Register 2

Address offset: 0x04
Reset value: 0x00007000

31-17	16	15	14	13	12	11-8
Reserved	RX IMMA START	RX START	LDMA TX	LDMA RX	FRXTH	DS [3:0]
rw	rw	rw	rw	rw	rw	rw

7	6	5	4	3	2	1	0
TXEIE	RXNEIE	ERRIR	FRF	NSSP	SSOE	TXDMA EN	RXDMA EN
rw	rw	rw	rw	rw	rw	rw	rw

Table 2: SPI CR2

Bits [31:17]
Reserved
Must be kept at reset value

Bit 16 RX IMMA START
Set this bit to receive data immediately after transmission finished.

This bit should be set before configuring the control register 1.
0: Immediate Receive data disabled
1: Immediate Receive data enabled
Note This bit is custom added to spi

Bit 15 RX START

Set this bit to start receive data only. This is similar to RX ONLY bit in control register 1.

0: Receive only disabled
1: Receive only enabled
Note This bit is custom added to spi

Bit 14 LDMA TX

Last DMA transfer for transmission.
This bit is used in data packing mode, to define if the total number of data to transmit by DMA is odd or even. It has significance only
if the TXDMAEN bit in the SPIx CR2 register is set and if packing mode is used. It has to be written when the SPI is disabled (SPE = 0 in the SPIx CR1 register)
0: Number of data to transfer is even
1: Number of data to transfer is odd

Bit 13 LDMA RX

Last DMA transfer for reception
This bit is used in data packing mode, to define if the total number of data to transmit by DMA is odd or even. It has significance only if the TXDMAEN bit in the SPIx CR2 register is set and if packingmode is used. It has to be written when the SPI is disabled (SPE=0 in the SPIx CR1 register
0: Number of data to transfer is even
1: Number of data to transfer is odd

Bit 12 FRXTH

FIFO reception threshold bit is used to set the threshold of the RX-FIFO that triggers an RXNE event.
0: RXNE event is generated if the FIFO level is greater than or equal to 1/2 (16-bit)
1: RXNE event is generated if the FIFO level is greater than or equal to 1/4 (8-bit)

Bits [11:8] DS [3:0]

These bits configure the data length for SPI transfers:
0000: Not used
0001: Not used
0010: Not used
0011: 4-bit
0100: 5-bit
0101: 6-bit
0110: 7-bit
0111: 8-bit
1000: 9-bit
1001: 10-bit
1010: 11-bit
1011: 12-bit
1100: 13-bit

1101: 14-bit
1110: 15-bit
1111: 16-bit
If software attempts to write one of the “Not used” values, they areforced to the value “0111”(8-bit).
Note: In place of using DS[3:0], we are using the TOTAL BITS TX for transmission and TOTAL BITS RX for reception which give us control over bits to be transferred and receive.

Bit 7 TXEIE
Tx buffer empty interrupt enable.
0: TXE interrupt masked
1: TXE interrupt not masked.
Used to generate an interrupt request when the TXE flag is set.

Bit 6 RXNEIE
RX buffer not empty interrupt enable.
0: RXNE interrupt masked.
1: RXNE interrupt not masked.
Used to generate an interrupt request when the RXNE flag is set.

Bit 5 ERRIR
Error interrupt enable bit controls the generation of interrupt when an error condition occurs. (CRCERR, OVR, MODF in SPI mode, FRE at TI mode).
0: Error interrupt is masked
1: Error interrupt is enabled

Bit 4 FRF
Frame Format
0: SPI Motorola mode
1: SPI TI mode
Note This bit must be written only when the SPI is disabled (SPE=0).

Bit 3 NSSP
NSS Pulse management bit is used in master mode only. it allows the SPI to generate an NSS pulse between two consecutive data when doing continuous transfers. In the case of a single data transfer, it forces the NSS pin high level after the transfer. It has no meaning ifCPHA = ’1’, or FRF = ’1’.
0: No NSS pulse
1: NSS pulse generated

Note This bit must be written only when the SPI is disabled (SPE=0).

Bit 2 SSOE
SS output enable
0: SS output is disabled in master mode and the SPI interface can work in a multi-master configuration
1: SS output is enabled in master mode and when the SPI interface is enabled. The SPI interface cannot work in a multi-master environment.

Bit 1 TXDMA EN
When Tx buffer DMA Enable bit is set, a DMA request is generated whenever the TXE flag is set.

0: Tx buffer DMA disabled
1: Tx buffer DMA enabled

Bit 0 RXDMA EN
When Rx buffer DMA enable bit is set, a DMA request is generated whenever the RXNE flag is set.
0: Rx buffer DMA disabled
1: Rx buffer DMA enabled

2.1.3 SPI Status Register

Address offset: 0x08
Reset value: 0x00000002

31-13	12-11	10-9	8	7	6	5	4	3-2	1	0
RSVD	FTLVL	FR LVL	FRE	BSY	OVR	MOD F	CRC ERR	RS VD	TXE	RXNE
r	r	r	r	r	r	r	r	r	r	r

Table 3: SPI SR

Bits [31:13] RSVD
Reserved, must be kept at reset value.

Bits[11:12] FTLVL
[1:0] FIFO Transmission Level. These bits are set and cleared by hardware.
00: FIFO empty
01: 1/4 FIFO
10: 1/2 FIFO
11: FIFO full (considered as FULL when the FIFO threshold is greater than 1/2)

Bits [9:10] FRLVL
[1:0] FIFO reception level.These bits are set and cleared by hardware.
00: FIFO empty
01: 1/4 FIFO
10: 1/2 FIFO
11: FIFO full

Bit 8 FRE
Frame format error. This flag is used for SPI in the TI slave mode. Refer to Section 38.4.11: SPI error flags. This flag is set by hardware and reset when SPIx SR is read by software.
0: No frame format error
1: A frame format error occurred

Bit 7 BSY
The BSY flag is set and cleared by hardware (writing to this flag has no effect). When BSY is set, it indicates that a data transfer is in progress on the SPI (the SPI bus is busy).
0: SPI not busy
1: SPI is busy in communication or Tx buffer is not empty

Bit 6 OVR
An overrun condition occurs when data is received by a master or slave and the RXFIFO has not enough space to store this received data.
0: No overrun occurred
1: Overrun occurred
When an overrun condition occurs, the newly received value does not overwrite the previous one in the RXFIFO. The newly received value is discarded and all data transmitted subsequently is lost

Bit 5 MODF
Mode fault occurs when the master device has its internal NSSsignal (NSS pin in NSS hardware mode, or SSI bit in NSS software mode) pulled low.
0: No mode fault occurred 1: Mode fault occurred
Bit 4 CRCERR
CRC error flag.
This flag is used to verify the validity of the value received when the CRCEN bit in the SPIx CR1 register isset.
0: CRC value received matches the SPIx RXCRCR value
1: CRC value received does not match the SPIx RXCRCR value

Bits [3:2] RSVD
[1:0] Reserved, must be kept at reset value.

Bit 1 TXE
Transmit buffer empty. This flag is set when transmission TX-FIFO has enough space to store data to send.
0: Tx buffer not empty
1: Tx buffer empty

Bit 0 RXNE
Receive buffer not empty. The RXNE flag is set depending on
the FRXTH bit value in the SPIx CR2 register.
0: Rx buffer empty
1: Rx buffer not empty

2.1.4 SPI Data Registers
There are total five data registers.

Data Registers	Address offset
SPIx_DR1	0x0C
SPIx_DR2	0x10
SPIx_DR3	0x14
SPIx_DR4	0x18
SPIx_DR5	0x1C

Bits [31:0] DR[31:0]
Data received or to be transmitted.The data register serves as an interface between the Rx and Tx FIFOs. When the dataregister is read, RxFIFO is accessed while the write to data register accesses TxFIFO.
To transfer data from register to FIFO, SPIx DR5 must have some data. When the data is received it goes to SPIx DR5 and the left shifted to DR4 and so on.

We don't use any the CRC calculation. So, following Registers are not used.

2.1.5 SPI CRC Polynomial Registers
Address offset: 0x20
Reset value: 0x0007
Bits [15:0] CRCPR [15:0]
CRC Polynomial register.
This register contains the polynomial for the CRC calculation.The CRC polynomial (0007h) is the reset value of this register. Another polynomial can beconfigured as required.
2.1.6 SPI Rx CRC register
Address offset: 0x24
Reset value: 0x0000
Bits[15:0] RXCRC [15:0]
Rx CRC register.
When CRC calculation is enabled, theRxCRC[15:0] bits contain the computed CRC value of the subsequently received bytes. This register is reset when the CRCEN bit in SPIx CR1 register is written to 1.
2.1.7 SPI Tx CRC register
Address offset: 0x28
Reset value: 0x0000
Bits [15:0] TXCRC [15:0]
Tx CRC Register. When CRC calculation is enabled, the TxCRC[7:0] bits contain the computed CRC value of the sub-sequently transmitted bytes. This register is reset when the CRCEN bit of SPIx-CR1 is written to 1.
2.2 The memory map for the SPI registers is shown in Table

Register Address	Data Width	Permission	Description
0x00020000	4 bytes	RW	SPI Configuration Register 1
0x00020004	4 bytes	RW	SPI Configuration Register 2
0x00020008	4 bytes	RO	SPI Status Register
0x0002000C	4 bytes	RW	SPI Data Register 1
0x00020010	4 bytes	RW	SPI Data Register 2
0x00020014	4 bytes	RW	SPI Data Register 3
0x00020018	4 bytes	RW	SPI Data Register 4
0x0002001C	4 bytes	RW	SPI Data Register 5
0x00020020	4 bytes	RW	SPI CRC polynomial register
0x00020024	4 bytes	RO	SPI Rx CRC register
0x00020028	4 bytes	RO	SPI Tx CRC register

Table4: SPI register memory map

3.Verification Overview

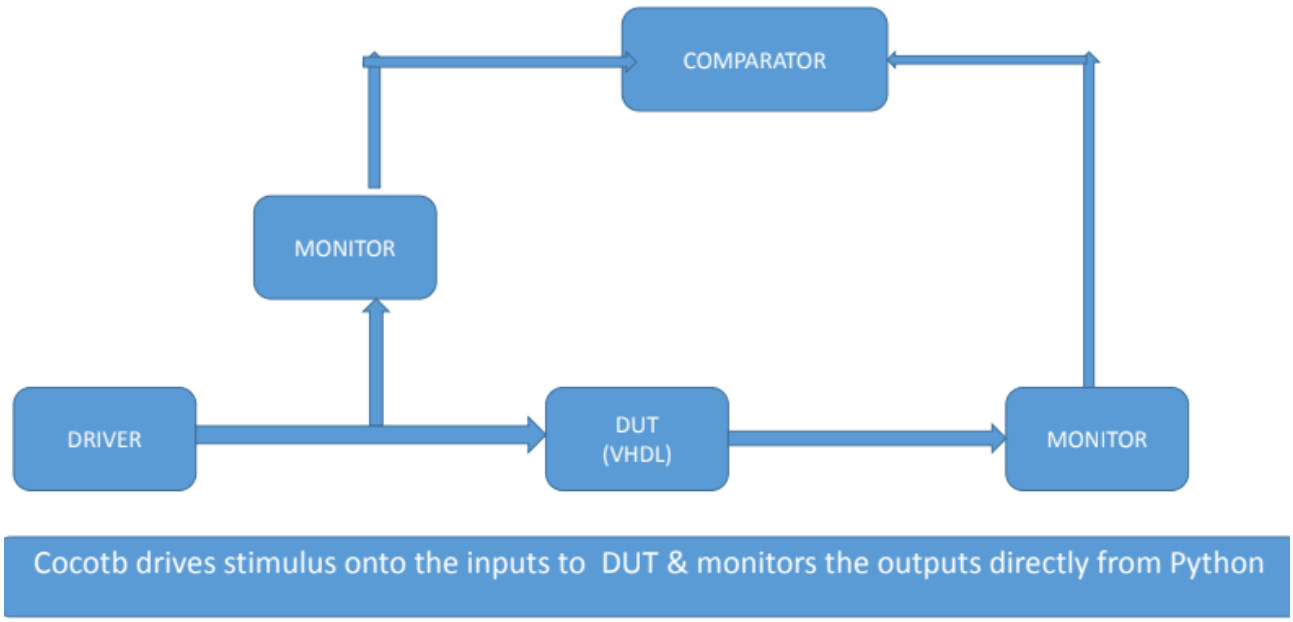


Fig. 2 SPI Verfication Overview

3.1.1 Driver

Successfully wrote cocotb based Python script for sending Data through AXI4 Lite Bus

• It communicates with the DUT (Serial Peripheral Interface (SPI)) in signal level.

• It is an active entity that decides how to drive signals to a particular interface of the design after recieving the sequences from the Sequencer.

• After finish driving to design, driver sends the response and gets ready for the next sequence from Sequencer

3.1.2 Test Bench

Test Bench is written in Python to Write/Read Data on SPI Register which is being send through AXI4 Lite Bus.

3.1.3 DUT

Verilog Design of SPI is being tested whether the correct data is successfully sent or not by seeing log file/ GTK wave or using Scoreboard.

3.2 Successful write on SPI

3.2.1 - Data 0x23 is send though test bench is verified on MOSI pin as in GTK wave

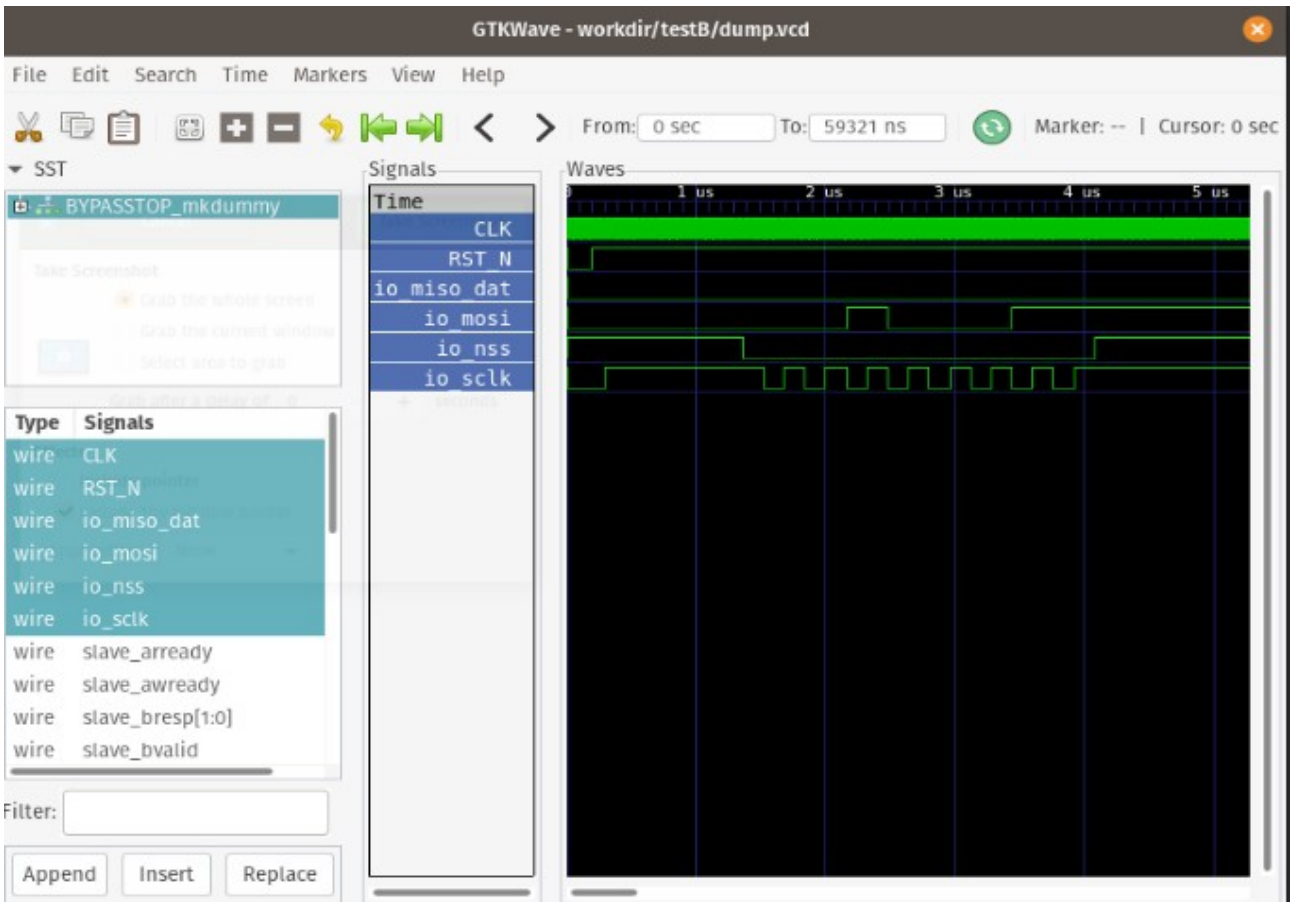


Fig. 3 GTK wave of MOSI pin, output of Data 0x23

3.2.2 - Multiple Data is send though test bench is verified MOSI pin as in GTK wave

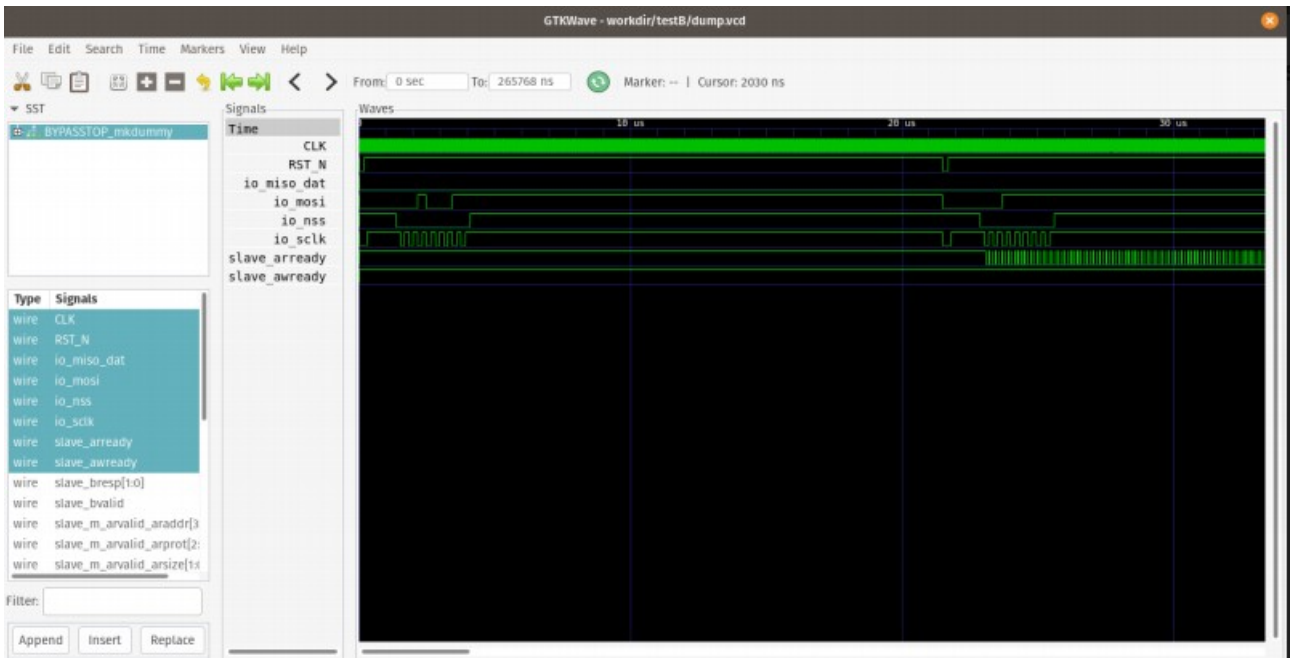


Fig. 4 GTK wave of MOSI pin, output of multiple Data sent to SPI

3.3 Verification of SPI

SPI is being verified manually by sending Data with the help of Drivers by writting Test Bench and checking out whether the data is being sent correct or not.

4. Verification of SPI interfaced with SRAM acting as slave

DUT (SPI) is interfaced with SRAM by creating wrapper file in verilog.

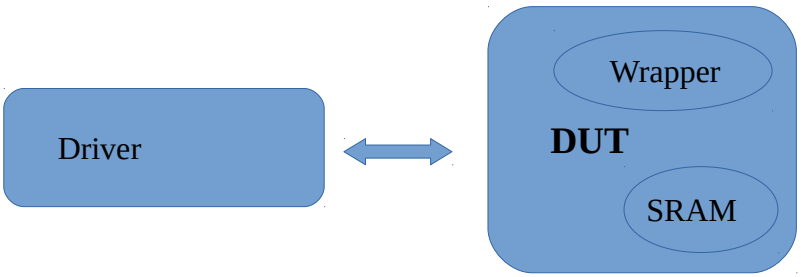


Fig. 5 SPI interfaced with SRAM

4.1.1 DUT Interfaced with SRAM Verification

The mail goal of Project is to verify the SPI designed which is interfaced with SRAM which is acting as slave to the SPI, SPI being act as here master.

4.1.2 SRAM Specification

Description

The ISSI IS62/65WVS5128GALL/GBLL are 4M bit Fast Serial static RAMs organized as 512K bytes by 8 bits. It is a dual die stack of two 2Mb Serial SRAMs.

The device is accessed via a simple Serial Peripheral Interface (SPI) compatible serial bus. The bus signals required are a clock input (SCK) plus separate data in (SI) and data out (SO) lines. Access the device is controlled through a Chip Select (CS#) input. Additionally, SDI (Serial Dual Interface) and SQI (Serial Quad Interface) is supported if your application needs faster data rates.This device also supports unlimited reads and writes to the memory array. The IS62/65WVS5128GALL/GBLL are available in the standard packages including 8-pin SOIC, 8-pin TSSOP and 24-ball TFBGA (6mm x 8mm).

4.1.3 Block Diagram of SRAM

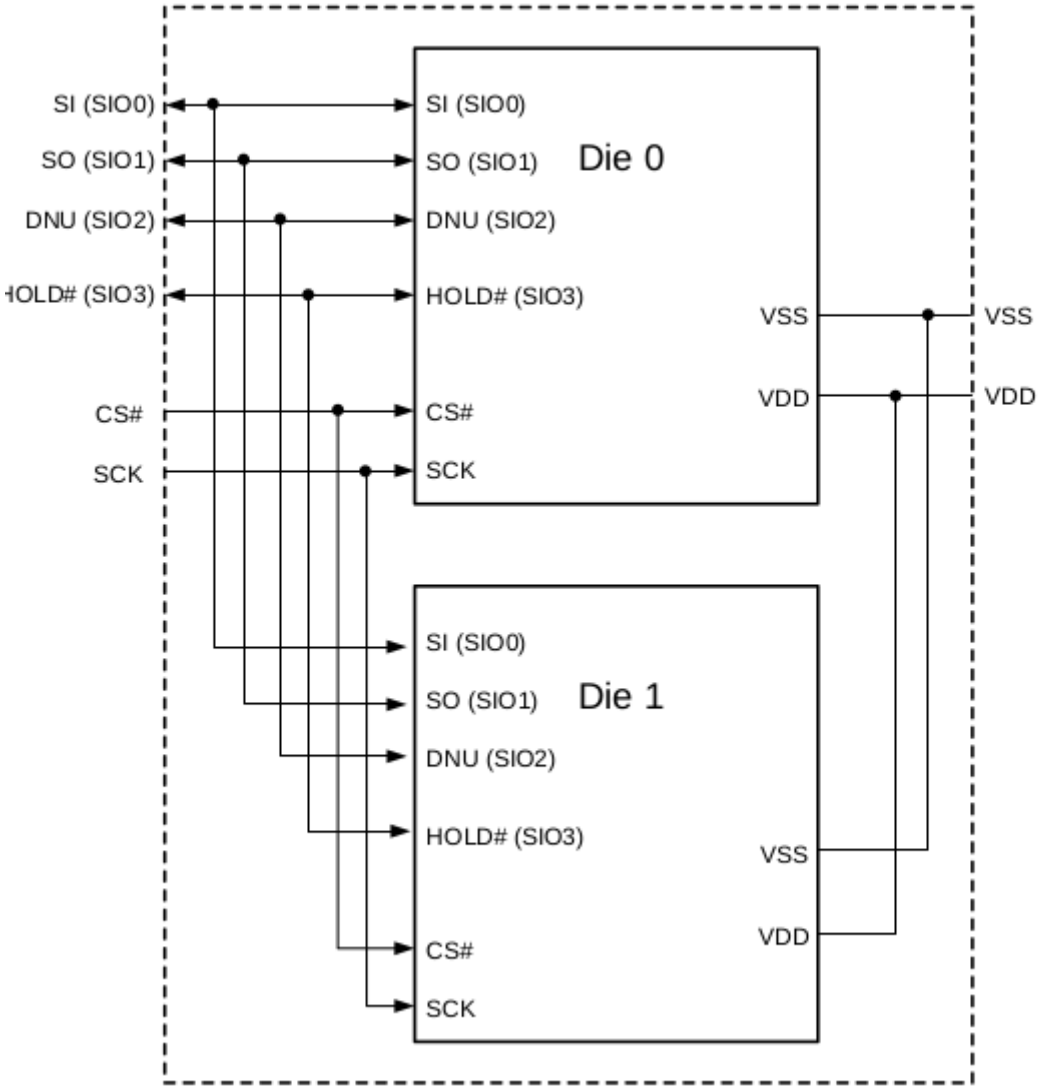
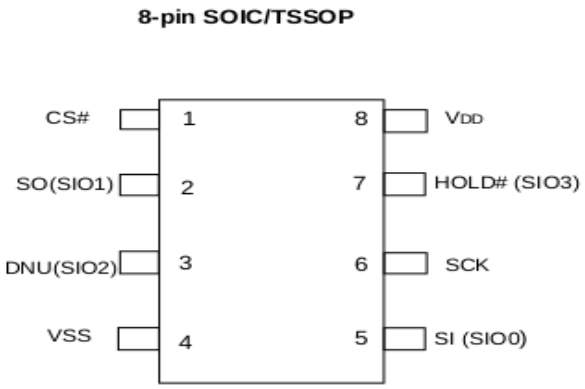


Fig. 6 SRAM IS62/65WVS5128GALL block digram

4.1.4 Pin Confugration of SRAM



PIN DESCRIPTIONS

CS#	Chip Enable Input
SO/SIO1	Serial Output/SIO1
DNU/SIO2	Do Not Use/SIO2
VSS	Ground
SI/SIO0	Serial Input/SIO0
SCK	Serial Clock
HOLD#/SIO3	HOLD#/SIO3
VDD	Power

Fig. 7 SRAM IS62/65WVS5128GALL pin confugration

4.1.5 Description of Pin of SRAM

Chip Select (CS#)

A low level on this pin selects the device. A high level deselects the device and forces it into Standby mode. When the device is deselected, SO goes to the high- impedance state, allowing multiple parts to share the same SPI bus. After power-up, a low level on CS# is required, prior to any sequence being initiated.

Serial Clock (SCK)

The SCK is used to synchronize the communication between a master and Serial SRAM. Instructions, addresses or data present on the SI pin are latched on the rising edge of the clock input, while data on the SO pin is updated after the falling edge of the clock input.

Serial Output (SO: SPI mode)

The SO pin is used to transfer data out of the device. During a read cycle, data is shifted out on this pin after the falling edge of the serial clock.

Serial Input (SI: SPI mode)

The SI pin is used to transfer data into the device. It receives instructions, addresses, and data. Data is latched on the rising edge of the serial clock.

HOLD Function (HOLD#: SPI Mode, and SDI Mode)

The HOLD# pin is used to suspend transmission to Serial SRAM while in the middle of a serial sequence without having to re-transmit the entire sequence over again. It must be held high any time this function is not being used. Once the device is selected and a serial sequence is underway, the HOLD# pin may be pulled low to pause further serial communication without resetting the serial sequence. The HOLD# pin should be brought low while SCK is low, otherwise the HOLD function will not be invoked until the next SCK high-to-low transition. The device must remain

selected during this sequence. The SI and SCK levels are “don’t cares” during the time the device is paused and any transitions on these pins will be ignored. To resume serial communication, HOLD# should be brought high while the SCK pin is low, otherwise serial communication will not be resumed until the next SCK high-to-low transition. The SO line will tri-state immediately upon a high-to low transition of the HOLD# pin, and will begin outputting again
Immediately upon a subsequent low- to-high transition of the HOLD pin, independent of the state of SCK.
Hold functionality is not available when operating in Quad SPI mode

Serial Input / Output Pins (SIO0, SIO1: SDI Mode)

The SIO0 and SIO1 pins are used for Dual SPI mode of operation (SI SIO0, SO-->SIO1). Functionality of these I/O pins is shared
with SO and SI.

Serial Input / Output Pins (SIO0, SIO1, SIO2, SIO3: SQI Mode)

The SIO0 through SIO3 pins are used for Quad SPI mode of operation. Because of the shared functionality of these pins the HOLD# feature is not available when using Quad SPI mode (Hold# SIO3 in Quad SPI mode)

DNU/SIO2 (Do Not Use or SIO2)

Pin 3 is DNU (Do No Use) in SPI mode and SDI mode. SIO3 in SQI mode.

FUNCTION DESCRIPTION

Serial SRAM is designed to interface directly with the Serial Peripheral Interface (SPI). It can also interface with Multi IO SPI interface (Dual SPI and Quad SPI).
The device contains an 8-bit instruction register. The device is accessed via the SI pin, with data being clocked in on the rising edge of SCK. The CS# pin must be low for the entire operation.
All instructions, addresses and data are transferred MSB first, LSB last.

OPERATION MODE

The device has three modes of operation that are selected by setting bits 7 and 6 in the MODE register. The modes of operation are Byte, Page and Sequential.

Byte Operation

Byte Operation is selected when bits 7 and 6 in the MODE register are set to 00. In this mode, the **read/ write operations**

are limited to only one byte. The Command followed by the 24-bit address is clocked into the device and the data to/from the device is transferred on the next eight clocks.

Page Operation

Page Operation is selected when bits 7 and 6 in the MODE register are set to 10. The device has 16,384 pages of 32 bytes. In this mode, the read and write operations are limited to within the addressed page (the address is automatically incremented internally). If the data being read or written reaches the page boundary, then the internal address counter will increment to the start of the page.

Sequential Operation

Sequential Operation is selected when bits 7 and 6 in the MODE register are set to 01. Sequential operation allows the entire array to be written to and read from. The internal address counter is automatically incremented until reaches the end of die boundary.The device is stacked with 2-die, so it has a restriction in sequential operation: The address counter cannot cross the die boundary. When the Address Pointer reaches the highest address of first die (3FFFFh), the address counter cannot cross to first address of 2 nd die (40000h). Instead, it rolls over to (00000h). So the sequential operation must be terminated at the last address of first die (Die 0) by CS# HIGH, and begin new sequential operation from first address (40000h) of second die (Die 1) by CS# LOW.

WRITE MODE

Prior to any attempt to write data to the device, the device must be selected by bringing CS# low. Once the device is selected, the Write command can be started by issuing a WRITE instruction, followed by the 24-bit address, with the first six MSB's of the address being a "don't care" bit, and then the data to be written. A write is terminated by the CS# being brought high.If operating in Page mode, after the initial data byte is shifted in, additional bytes can be shifted into the device. The Address Pointer is automatically incremented. This operation can continue for the entire page (32 bytes) before data will start to be overwritten.

If operating in Sequential mode, after the initial data byte is shifted in, additional bytes can be clocked into theDevice. The internal Address Pointer is automatically incremented until reaches to die boundary address.

READ MODE

The device is selected by pulling CS# Low. Then 8 bit instruction is transmitted to the device followed by the 24 bit address, with the first seven MSB's of the address being a "don't care" bit. After the correct READ instruction and address are sent, the data stored in the memory at the selected address is shifted out on the SO pin. If operating in Sequential mode, the data stored in the

memory at the next address can be read sequentially by continuing to provide clock pulses until reaches to die boundary address.

4.1.6 Instruction Set

Instruction Name	Instruction Format	Hex Code	Description
READ	0000 0011	0x03	Read data from memory array beginning at selected address
WRITE	0000 0010	0x02	Write data to memory array beginning at selected address
ESDI	0011 1011	0x3B	Enter SDI mode
ESQI	0011 1000	0x38	Enter SQI mode
RSTDQI	1111 1111	0xFF	Reset SDI/SQI mode
RDMR	0000 0101	0x05	Read Mode Register
WRMR	0000 0001	0x01	Write Mode Register

Fig. 8 Instruction set of SRAM IS62/65WVS5128GALL

4.1.7 BYTE READ OPERATION (SPI MODE)

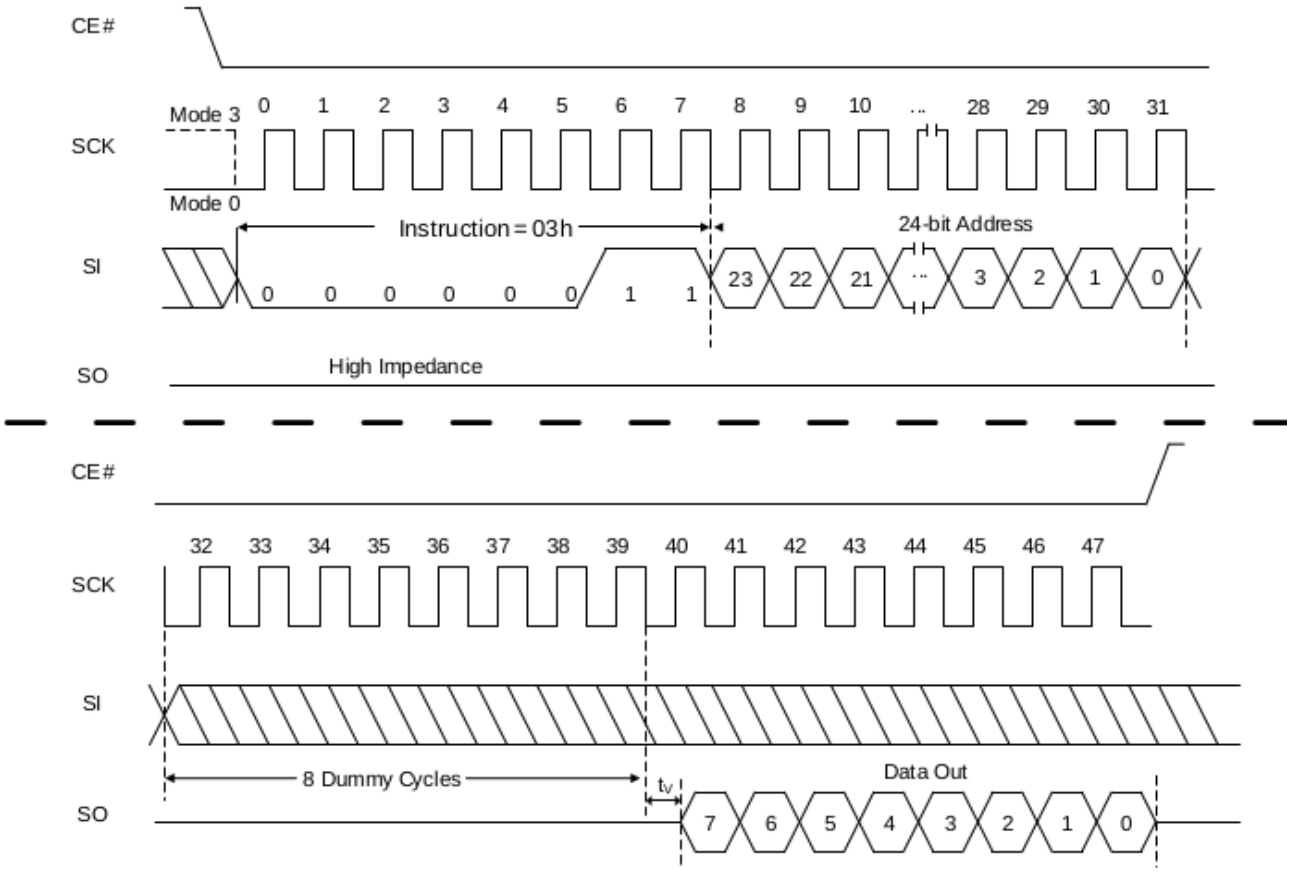


Fig. 9 Byte Read Operation of SRAM IS62/65WVS5128GALL

4.1.8 BYTE WRITE OPERATION (SPI MODE)

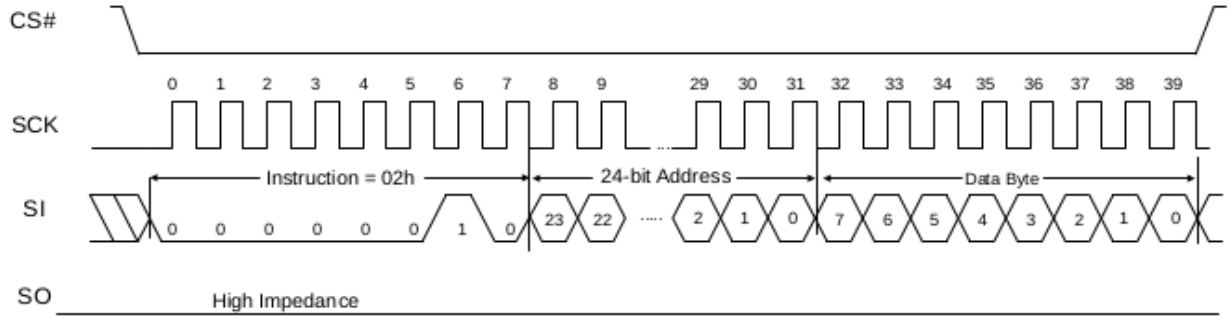


Fig. 10 Byte Write Operation of SRAM IS62/65WVS5128GALL

4.1.9 SRAM wrapper file

```
`timescale 1ns/10ps
module IS6265WVS5128GABLL_spi_sram (SI, SCK, CS_N, RESET, SO);

  input          SI;                  // serial data input/SIO0
  input          SCK;                   // serial data clock

  input          CS_N;                   // chip select - active low



  input          RESET;                   // model reset/power-on reset


  output          SO;                  // serial data output/SIO1

 IS6265WVS5128GABLL_SPI_only bfm(
            // Data inputs/outputs
            .SI(SI),
            .SCK(SCK),
            .CS_N(CS_N),
            .HOLD_N(1'b1),
            .RESET(RESET),
            .DNU(),
            .SO(SO) );
endmodule
```

Verification Process of SPI interfaced with SRAM

Procedure for writting Data on SRAM via SPI
1. Setting CR1 of SPI with value 0x0000003B to start SPI
2. We need to send the data to DR1 along with the address of SRAM where we want to Write the Data via SPI i.e. 0x0000003B
3.DR2 and DR3 of SPI is left with some zero value i.e 0x00000000
4.While Reading to the Registers we need to configure CR2 with 0x00000000 value, while writting we need not do anything on CR2.
5. to end the Writting on SPI we need to configure CR1 with 0x0028007B.

Data is being saved on SRAM with specified address.
For Verification of Data being Sent on SRAM via SPI is correct or not, Read the saved data on the same address and compare it whether it is equal or not.
Procedure
1. Confure DR1 with 0x03000080
2.DR2 and DR5 set to 0x00000000
3. Configure CR2 with 0x00010000
4.to end the reading process configure CR1 with 0x0828007B

Sample Test bench

```
import os
import sys
import cocotb
import logging
from cocotb.result import TestFailure
from cocotb.result import TestSuccess
from cocotb.clock import Clock
import time
from array import array as Array
from cocotb.triggers import Timer, ClockCycles
from AXI_Driver import AXI4LiteMaster
from AXI_Driver import AXIProtocolError

CLK_PERIOD = 20

'''
Addres of register
'''
CR1 = 0x00020000
CR2  =0x00020004
SR    =0x00020008
DR1   =0x0002000C
DR2   =0x00020010
DR3   =0x00020014
DR4   =0x00020018
DR5   =0x0002001C
CRCPR =0x00020020
```

```
RXCRCR=0x00020024
TXCRCR=0x00020028

def setup_dut(dut):
    cocotb.fork(Clock(dut.CLK, CLK_PERIOD).start())

@cocotb.test(skip = False)
def write_and_read(dut):
    """Write to the register at address 0.
    Read back from that register and verify the value is the same.

    Test ID: 1

    Expected Results:
        The contents of the register is the same as the value written.
    """

    # Reset
    dut.RST_N <= 0
    axim = AXI4LiteMaster(dut, None, dut.CLK)
    setup_dut(dut)
    yield Timer(CLK_PERIOD * 10)

    dut.RST_N <= 1

    # Write to the register
    yield axim.write(CR1, 0x0000003B)
    yield Timer(CLK_PERIOD * 10 )

    yield ClockCycles (dut.CLK,1000, True)


    # Write to the register
    yield axim.write(DR1, 0x02000080)
    yield Timer(CLK_PERIOD * 10 )

    # Write to the register
    yield axim.write(DR2, 0xaa000000)
    yield Timer(CLK_PERIOD * 10 )

    yield axim.write(DR5, 0x22000000)
    yield Timer(CLK_PERIOD * 10)

    #Read to the register while writting need not to define CR2
    yield axim.write(CR2, 0x00000000)
    yield Timer(CLK_PERIOD * 10 )

    # Write to the register
    yield axim.write(CR1, 0x0028007B)
    yield Timer(CLK_PERIOD * 10)
```

```
#dut.RST_N <= 0
#yield Timer(CLK_PERIOD * 10)
#dut.RST_N <= 1
yield ClockCycles (dut.CLK,1000, True)

# Write to the register
yield axim.write(DR1,0x03000080)
yield Timer(CLK_PERIOD * 10  )

# Write to the register
yield axim.write(DR2, 0x00000000)
yield Timer(CLK_PERIOD * 10 )

yield axim.write(DR5, 0x22000000)
yield Timer(CLK_PERIOD * 10)

#Read to the register while writting need not to define CR2
yield axim.write(CR2, 0x00000000)
yield Timer(CLK_PERIOD * 10 )

#Read to the register while writting need not to define CR2
yield axim.write(CR2, 0x00010000)
yield Timer(CLK_PERIOD * 10 )

# Write to the register
yield axim.write(CR1, 0x0828007B)
yield Timer(CLK_PERIOD * 10)

yield ClockCycles (dut.CLK,100, True)

# Read back the value
value = yield axim.read(DR5)
yield Timer(CLK_PERIOD * 10)
```

For verifying above steps, a log file is being generated on coctb-verilator tools

0xaa is written address 0x000100

the log file is generated below

```
 737135 SPI : Controller Write Channel
 737165 SPI : Write request wr_addr 0002000c wr_data 02000100
 741435 SPI : Controller Write Channel
 741465 SPI : Write request wr_addr 00020010 wr_data aa000000
 745435 SPI : Controller Write Channel
 745465 SPI : Write request wr_addr 0002001c wr_data 00000000
Error: "devices/spi/./spi.bsv", line 385, column 8: (R0002)
Conflict-free rules RL_spi_spi_rl_transmit_idle and
RL_spi_spi_rl_write_to_cfg called conflicting methods read and write of
module instance spi_spi_tx_data_en.
```

745475 SPI : DR to tx_fifo data 02 rg_data_counter 0
745485 SPI : DR to tx_fifo data 00 rg_data_counter 1
745495 SPI : DR to tx_fifo data 01 rg_data_counter 2
745505 SPI : DR to tx_fifo data 00 rg_data_counter 3
745515 SPI : DR to tx_fifo data aa rg_data_counter 4
745525 SPI : DR to tx_fifo data 99 rg_data_counter 5
745535 SPI : DR to tx_fifo data 55 rg_data_counter 6
745545 SPI : DR to tx_fifo data aa rg_data_counter 7
745555 SPI : DR to tx_fifo data 00 rg_data_counter 8
745565 SPI : DR to tx_fifo data 00 rg_data_counter 9
745575 SPI : DR to tx_fifo data 00 rg_data_counter 10
745585 SPI : DR to tx_fifo data 00 rg_data_counter 11
745595 SPI : DR to tx_fifo data 00 rg_data_counter 12
745605 SPI : DR to tx_fifo data 00 rg_data_counter 13
745615 SPI : DR to tx_fifo data 00 rg_data_counter 14
745625 SPI : DR to tx_fifo data 00 rg_data_counter 15
745635 SPI : DR to tx_fifo data 00 rg_data_counter 16
745645 SPI : DR to tx_fifo data 00 rg_data_counter 17
745655 SPI : DR to tx_fifo data 00 rg_data_counter 18
745665 SPI : DR to tx_fifo data 00 rg_data_counter 19
752155 SPI : Controller Write Channel
752185 SPI : Write request wr_addr 00020000 wr_data 0028007b
Error: "devices/spi/./spi.bsv", line 385, column 8: (R0002)
Conflict-free rules RL_spi_spi_rl_transmit_idle and
RL_spi_spi_rl_write_to_cfg called conflicting methods read and write of
module instance spi_spi_rg_spi_cfg_cr1.

752195 SPI : Transmit state has started rg_data_tx 02
752275 SPI : START_TRANSMIT case2 counter 00 data 0 rg_bit_count 00
752435 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 1
752595 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 2
752755 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 3
752915 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 4
753075 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 5
753235 SPI : DATA_TRANSMIT case 2 data 1 rg_data_counter 06 rg_bit_count 6
753395 SPI : tx_fifo data 00
753395 SPI : DATA_TRANSMIT case 3 data 0 rg_data_counter 07 rg_bit_count 7
753555 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 8
753715 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 9
753875 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 10
754035 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 11
754195 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 12
754355 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 13
754515 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 14
754675 SPI : tx_fifo data 01
754675 SPI : DATA_TRANSMIT case 3 data 0 rg_data_counter 07 rg_bit_count 15
754835 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 16
754995 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 17
755155 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 18
755315 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 19

```
 755475 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 20
 755635 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 21
 755795 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 22
 755955 SPI : tx_fifo data 00
 755955 SPI : DATA_TRANSMIT case 3 data 1 rg_data_counter 07 rg_bit_count 23
 756115 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 24
 756275 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 25
 756435 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 26
 756595 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 27
 756755 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 28
 756915 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 29
 757075 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 30
 757235 SPI : tx_fifo data aa
 757235 SPI : DATA_TRANSMIT case 3 data 0 rg_data_counter 07 rg_bit_count 31
 757395 SPI : DATA_TRANSMIT case 1 data 1 rg_data_counter 00 rg_bit_count 32
 757555 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 33
 757715 SPI : DATA_TRANSMIT case 1 data 1 rg_data_counter 02 rg_bit_count 34
 757875 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 35
 758035 SPI : DATA_TRANSMIT case 1 data 1 rg_data_counter 04 rg_bit_count 36
 758195 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 37
 758355 SPI : DATA_TRANSMIT case 2 data 1 rg_data_counter 06 rg_bit_count 38
 758515 SPI: ABORT tx 0
/--- Write (000100) = 10101010; ---/
 758675 SPI : Transmit state is in idle
1293865 SPI : Controller Write Channel
1293895 SPI : Write request wr_addr 0002000c wr_data 03000100
1297865 SPI : Controller Write Channel
1297895 SPI : Write request wr_addr 00020010 wr_data 00000000
1302435 SPI : Controller Write Channel
1302465 SPI : Write request wr_addr 0002001c wr_data 00000000
Error: "devices/spi/./spi.bsv", line 385, column 8: (R0002) Conflict-free rules
RL_spi_spi_rl_transmit_idle and RL_spi_spi_rl_write_to_cfg called conflicting methods read and
write of
 module instance spi_spi_tx_data_en.

1302475 SPI : DR to tx_fifo data 03 rg_data_counter  0
1302485 SPI : DR to tx_fifo data 00 rg_data_counter  1
1302495 SPI : DR to tx_fifo data 01 rg_data_counter  2
1302505 SPI : DR to tx_fifo data 00 rg_data_counter  3
1302515 SPI : DR to tx_fifo data 00 rg_data_counter  4
1302525 SPI : DR to tx_fifo data 00 rg_data_counter  5
1302535 SPI : DR to tx_fifo data 00 rg_data_counter  6
1302545 SPI : DR to tx_fifo data 00 rg_data_counter  7
1302555 SPI : DR to tx_fifo data 00 rg_data_counter  8
1302565 SPI : DR to tx_fifo data 00 rg_data_counter  9
1302575 SPI : DR to tx_fifo data 00 rg_data_counter 10
1302585 SPI : DR to tx_fifo data 00 rg_data_counter 11
1302595 SPI : DR to tx_fifo data 00 rg_data_counter 12
1302605 SPI : DR to tx_fifo data 00 rg_data_counter 13
1302615 SPI : DR to tx_fifo data 00 rg_data_counter 14
```

1302625 SPI : DR to tx_fifo data 00 rg_data_counter 15
1302635 SPI : DR to tx_fifo data 00 rg_data_counter 16
1302645 SPI : DR to tx_fifo data 00 rg_data_counter 17
1302655 SPI : DR to tx_fifo data 00 rg_data_counter 18
1302665 SPI : DR to tx_fifo data 00 rg_data_counter 19
1307715 SPI : Controller Write Channel
1307745 SPI : Write request wr_addr 00020004 wr_data 00010000
Error: "devices/spi/./spi.bsv", line 385, column 8: (R0002)
 Conflict-free rules RL_spi_spi_rl_transmit_idle and
 RL_spi_spi_rl_write_to_cfg called conflicting methods read and write of
 module instance spi_spi_rg_spi_cfg_cr2.

1315465 SPI : Controller Write Channel
1315495 SPI : Write request wr_addr 00020000 wr_data 0828007b
Error: "devices/spi/./spi.bsv", line 385, column 8: (R0002)
 Conflict-free rules RL_spi_spi_rl_transmit_idle and
 RL_spi_spi_rl_write_to_cfg called conflicting methods read and write of
 module instance spi_spi_rg_spi_cfg_cr1.

1315505 SPI : Transmit state has started rg_data_tx 03
1315585 SPI : START_TRANSMIT case2 counter 00 data 0 rg_bit_count 00
1315745 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 1
1315905 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 2
1316065 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 3
1316225 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 4
1316385 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 5
1316545 SPI : DATA_TRANSMIT case 2 data 1 rg_data_counter 06 rg_bit_count 6
1316705 SPI : tx_fifo data 00
1316705 SPI : DATA_TRANSMIT case 3 data 1 rg_data_counter 07 rg_bit_count 7
1316865 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 8
1317025 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 9
1317185 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 10
1317345 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 11
1317505 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 12
1317665 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 13
1317825 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 14
1317985 SPI : tx_fifo data 01
1317985 SPI : DATA_TRANSMIT case 3 data 0 rg_data_counter 07 rg_bit_count 15
1318145 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 16
1318305 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 17
1318465 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 18
1318625 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 19
1318785 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 20
1318945 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 21
1319105 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 22
1319265 SPI : tx_fifo data 00
1319265 SPI : DATA_TRANSMIT case 3 data 1 rg_data_counter 07 rg_bit_count 23
1319425 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 24
1319585 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 25
1319745 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 26

1319905 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 27
1320065 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 28
1320225 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 29
1320385 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 30
1320545 SPI : tx_fifo data 00
1320545 SPI : DATA_TRANSMIT case 3 data 0 rg_data_counter 07 rg_bit_count 31
1320705 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 00 rg_bit_count 32
1320865 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 01 rg_bit_count 33
1321025 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 02 rg_bit_count 34
1321185 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 03 rg_bit_count 35
1321345 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 04 rg_bit_count 36
1321505 SPI : DATA_TRANSMIT case 1 data 0 rg_data_counter 05 rg_bit_count 37
1321665 SPI : DATA_TRANSMIT case 2 data 0 rg_data_counter 06 rg_bit_count 38
1321825 SPI: ABORT tx 0
1321905 SPI : Receive has started
/--- Read (000100) = 10101010; ---/
1322065 SPI : START_RECEIVE case2 counter 00
1322225 SPI : DATA_RECEIVE case1 counter 01 data_rx 02 rg_bit_count 1
1322385 SPI : DATA_RECEIVE case1 counter 02 data_rx 05 rg_bit_count 2
1322545 SPI : DATA_RECEIVE case1 counter 03 data_rx 0a rg_bit_count 3
1322705 SPI : DATA_RECEIVE case1 counter 04 data_rx 15 rg_bit_count 4
1322865 SPI : DATA_RECEIVE case1 counter 05 data_rx 2a rg_bit_count 5
1323025 SPI : DATA_RECEIVE case1 counter 06 data_rx 55 rg_bit_count 6
1323185 SPI: ABORT rg_data_rx aa
1323195 SPI : rx_fifo to dr reg 00000000000000000000000000000000
1323265 SPI : Transmit state is in idle
2216645 SPI : Controller Write Channel
2216675 SPI : Write request wr_addr 0002000c wr_data 00000001

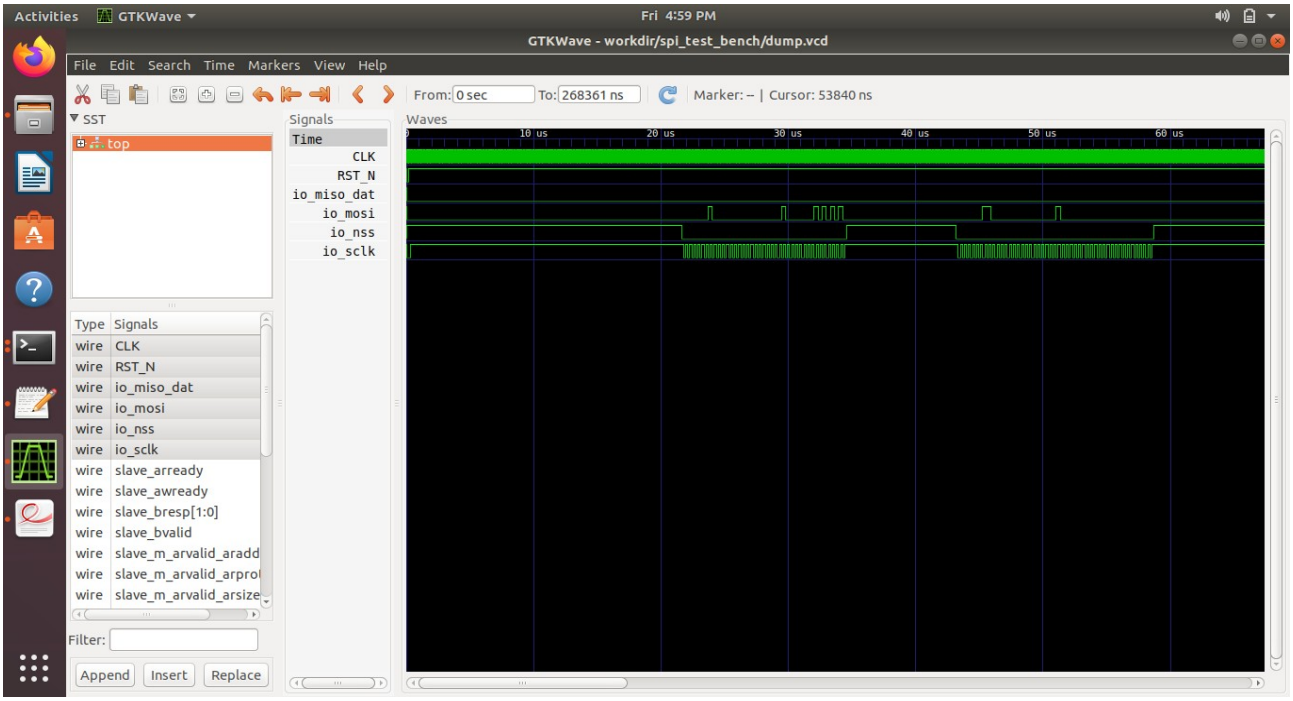


Fig. 11 Successful 0xaa Byte Write Operation of SRAM IS62/65WVS5128GALL as shown on MOSI Pin

4.1.10 Conclusion

- SPI Design is successfully verified by checking the Output pin MOSI by comparing the sent data and ouput data of MOSI pin.
- SPI Design interfaced with SRAM acting as Slave is also verified by writting Data on SRAM which is sent through SPI and the same data that was written on specified address is being read and compared which sent data, if Data is same then we cansay Designed is working perfectly otherwise there is issue in design.

4.1.11 Future Work

Although many milestones and achievements were reached in this project, this is not a perfect verification testbench written, an efficient testbench can be wriiten that can be re-usable test bench. Future work include writting script of Coverage , by seeing the coverage report we can test all the possible case by sending data through test bench, how many combination of data is being sent and tested correctly.

5. References

1. https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
2. https://docs.cocotb.org/en/latest/quickstart.html
3. http://www.issi.com/WW/pdf/IS62-65WVS5128GALL-BLL.pdf